Development of a GPU-accelerated MIKE 21 Solver for Water Wave Dynamics

Peter Edward Aackermann, s093066

Peter J. Dinesen Pedersen, s093053

IMM-B.Sc.-2012-29

Kgs. Lyngby July 27, 2012

Supervisor Associate Professor Allan P. Engsig-Karup
Department of Informatics and Mathematical Modelling, IMM

Supervisor Thomas Clausen and Jesper Grooss
DHI Group, Hørsholm, Denmark

B.Sc. in Mathematics and Technology
Technical University of Denmark

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Sc.-2012-29

Abstract (English)

Development of a GPU-accelerated MIKE 21 Solver for Water Wave Dynamics

With encouragement by the company DHI are the aim of this B.Sc. thesis to investigate, whether if it is possible to accelerate the simulation speed of DHIs commercial product MIKE 21 HD, by formulating a parallel solution scheme and implementing it to be executed on a CUDA-enabled GPU (massive parallel hardware).

MIKE 21 HD is a simulation tool, which simulates water wave dynamics in lakes, bays, coastal areas and seas by solving a set of hyperbolic partial differential equations called shallow water equations. The solution scheme is the Alternating Direction Implicit (ADI) method, which results in a lot of tri-diagonal matrix systems, which have to be solved efficiently.

Two different parallel solution schemes are implemented. The first (`S1`) solves each tri-diagonal in parallel using a single CUDA thread for each system. This approach use the same solution algorithm as MIKE 21 HD, Thomas algorithm. The other solution schemes (`S2`) adds more parallelism into the system by using several threads to solve each system in parallel. In order to do this efficient are several parallel solution algorithms investigated. The focus have been on the Parallel Cyclic Reduction (PCR) algorithm and a hybrid algorithm of Cyclic Reduction (CR) and PCR.

We discover that `S2` are beneficial to use for small problems, while `S1` yields better results for larger systems. We have obtained 42x and 80x speedup in double-precision for `S1` and `S2` respectively, compared to a representative sequential C implementation of MIKE 21 HD. Furthermore, the impact of switching to perform calculation in single-precision been investigated. This resulted in 145x and 203x speedup for `S1` and `S2`, respectively. However, this had some precision lost when using single-precision. All test throughout the project is performed on the graphics card NVIDIA GeForce GTX 590.

Resumé (Danish)

Udvikling af en GPU-accelereret MIKE 21 løser for vandbølgedynamik

Formålet med denne afhandling er på opfordring af virksomheden DHI, at undersøge, hvorvidt det er muligt at accelerere simuleringshastigheden af DHI's kommercielle produkt MIKE 21 HD, ved at formulere en parallel løsningsmetode og implementere denne på en CUDA-enabled GPU (massivt parallel hardware).

MIKE 21 HD er et simuleringsværktøj, der simulerer vandbølgedynamik i søer, bugter, kystområder og hav ved at løse hyperbolske partielle differentialligninger kendt som lavvande ligningerne (shallow water equations). Løsningsmetoden der avendes er kendt som Alternating Direction Implicit (ADI), hvilket resulterer i et stort antal tri-diagonale matrix systemer der skal løses effektivt.

Der er implementeret to parallelle løsningsmetoder. Den ene (`S1`) løser hvert tri-diagonal system parallelt ved brug af én CUDA tråd per system. Her anvendes samme løsningsalgoritme som i MIKE 21 HD, Thomas algoritmen. Den anden metode (`S2`) tilføjer yderligere parallelitet i systemet ved at lade flere tråde løse hvert system parallelt. For at gøre dette effektivt, er parallelle tri-diagonale løsningsmetoder blevet udforsket. Der er tager udgangspunkt i algoritmerne Parallel Cyclic Reduction (PCR) og en hybrid algoritme imellem Cyclic Reduction (CR) og PCR.

Vi har fundet frem til at `S2` er fordelagtig for små problemstørrelser, mens `S1` giver gode resultater for større problemer. Der er opnået 42x og 80x speedup for henholdsvis `S1` og `S2` i double præcision, sammenlignet med en repræsentativ sekventiel C implementering af MIKE 21 HD. Det er desuden undersøgt, hvordan køretiden influeres ved at udføre beregningerne i single præcision. Dette resultere i helt op til 145x og 203x hurtigere end den sekventielle C applikation. Dog med en reduktion i nøjagtighed. Alle test igennem projektet er udført på grafikkortet NVIDIA GeForce GTX 590.

Preface

This thesis is submitted as a partial fulfilment of the requirements for obtaining the Danish Bachelor of Science degree at the Technical University of Denmark (DTU). The work has been carried out partially at DHI Group in Hørsholm with supervisor Thomas Clausen and Jesper Grooss and partially at the Department of Informatics and Mathematical Modelling, DTU, under supervision of Associate Professor Allan Peter Engsig-Karup. The project started February 6$^{\text{th}}$ 2012 and was completed July 27$^{\text{th}}$ 2012, having a workload of 15 ECTS points per author.

The thesis investigates the possibility to improve the simulation speed by developing a parallel solution scheme to accelerate MIKE 21 HD; a solver for water wave dynamics developed by DHI Hørsholm. We chose this project, because it had potential to be a very challenging and ambitious project demanding knowledge obtained in courses, but also provided challenges to which we had no prior knowledge. The project proved to be very challenging and far more complex than ever experienced before. Both of us are very curious in general and find non-trivial problems highly interesting, thus this thesis turned out to be exactly what we had hoped for.

The thesis consists of this report and a source code booklet, where the sequential and parallel implementations of MIKE 21 HD are listed.

Kgs. Lyngby, July 27$^{\text{th}}$, 2012

Peter Edward Aackermann
s093066

Peter J. Dinesen Pedersen
s093053

Acknowledgements

We would like to thank our supervisor Allan Peter Engsig-Karup for great help and inspiration throughout the project and for always being at our disposal, when questions occurred. We would also like to thank Thomas Clausen and Jesper Grooss for always being helpful and for repeatedly executing tests on MIKE 21 HD when comparison was needed.

We are truly grateful for this.

Declarations

We have participated at GRØN DYST at DTU[1], where we presented the work in this thesis. GRØN DYST is a student conference on sustainability, climate technology and the environment. The presentation of our project was well received and the judging panel showed great interest in the project and our work.

[1] More information see http://www.groendyst.dtu.dk/English.aspx

Contents

Abstract (English) i

Resumé (Danish) ii

Preface iii

Acknowledgements iv

Declarations v

1 Introduction 1
- 1.1 Validation of Scientific Computing . . . 3
- 1.2 Proposal from DHI . . . 4
- 1.3 Thesis Statement . . . 5
- 1.4 Scope of the Project . . . 6
- 1.5 Learning Objectives . . . 7
- 1.6 Thesis Structure . . . 8

2 Scalability and Expectations of Speedup 9
- 2.1 Strong Scaling . . . 9
- 2.2 Weak Scaling . . . 11
- 2.3 Applying Strong and Weak Scaling . . . 11

3 CUDA Theory 12
- 3.1 Processing Flow for GPGPU . . . 13
- 3.2 CUDA C . . . 14
- 3.3 Fermi Architecture . . . 15
- 3.4 Data Transfer Between Host and Device . . . 16
- 3.5 Global Memory . . . 17
  - 3.5.1 Coalesced memory access . . . 17
- 3.6 L1 Cache and Shared Memory . . . 18
  - 3.6.1 Shared memory bank conflicts . . . 18

3.7 Registers . . . 19
3.7.1 Register spilling . . . 19
3.8 Latency Hiding . . . 19
3.9 Occupancy . . . 20
3.10 `nvcc` Compiler . . . 20
3.11 NVIDIA Visual Profiler . . . 21
3.12 Calculation of Performance Metrics . . . 22
3.12.1 Identifying performance limiters: Memory or compute bound . . . 22
3.12.2 Theoretical bandwidth calculation . . . 23
3.12.3 Effective bandwidth . . . 23
3.12.4 Divergent branches . . . 23
3.12.5 Control flow divergence . . . 23
3.12.6 Replayed instructions . . . 23
3.13 Performance Optimization Strategies . . . 24

4 Numerical Formulation **25**
4.1 Shallow Water Equations . . . 25
4.2 Introduction to Derivation of Discretization . . . 27
4.2.1 Methods for discretization and solution scheme . . . 27
4.2.2 Staggered grid in (x, y)-space . . . 28
4.2.3 Time centering . . . 29
4.3 Discretization of Mass Equations . . . 30
4.4 Discretization of Momentum Equations . . . 31
4.4.1 Discretization of the time derivation term . . . 31
4.4.2 Discretization of the convective momentum term . . . 31
4.4.3 Discretization of the cross momentum term . . . 33
4.4.4 Discretization of the gravity term . . . 34
4.4.5 Discretization of the resistance term . . . 34
4.5 Setting Coefficients for an x-sweep . . . 35
4.5.1 Coefficient for mass equation . . . 35
4.5.2 Coefficient for momentum equation . . . 36
4.5.3 Set up the penta-diagonal matrix system . . . 38

5 Tri-diagonal Solver Algorithms **39**
5.1 Tri-diagonal matrix . . . 39
5.2 The Thomas Algorithm . . . 40
5.2.1 Forward elimination . . . 40
5.2.2 Backwards substitution . . . 41
5.3 Parallel Cyclic Reduction . . . 42
5.4 Cyclic Reduction + Parallel Cyclic Reduction Hybrid . . . 45

6 Sequential C Implementation **48**
6.1 Comprehension into MIKE 21 HD . . . 48
6.1.1 Outlining the MIKE 21 HD flow operate . . . 49
6.2 Data Structures . . . 49
6.3 Development of the Sequential C Implementation . . . 51
6.3.1 Implementation details . . . 51
6.3.2 Complexity of C application . . . 53
6.4 Handling of Boundary Conditions . . . 53
6.5 Investigating Performance of MIKE 21 HD and C Application . . . 58
6.5.1 Profiling MIKE 21 HD . . . 58
6.5.2 Profiling the C implementation . . . 59
6.5.3 Performance comparison of MIKE 21 HD and C implementation . . . 59
6.6 Verification . . . 60
6.7 Validation . . . 61

7 Parallel CUDA C Implementation **65**
7.1 Parallel Approaches . . . 66
7.2 Optimization Strategy . . . 67
7.2.1 Compute vs. memory bound . . . 67
7.3 Test Environment . . . 68
7.3.1 Test configurations . . . 69
7.3.2 Time measuring . . . 70
7.3.3 Execution safety . . . 70
7.3.4 Validation . . . 70

8 CUDA C Optimization **71**
8.1 Data Transfer Between Host and Device . . . 71
8.2 Method 1 . . . 73
8.2.1 Naive . . . 73
8.2.2 Naive version 2 . . . 74
8.2.3 Version 2 . . . 78
8.2.4 Version 3 . . . 83
8.2.5 Version 4 . . . 89
8.3 Method 2 . . . 95
8.3.1 Naive version . . . 95
8.3.2 Version 2 . . . 97
8.3.3 Version 3 . . . 100
8.3.4 Version 4 . . . 101
8.4 Solver . . . 105
8.4.1 Parallel cyclic reduction . . . 105
8.4.2 Cyclic reduction + parallel cyclic reduction hybrid . . . 107
8.4.3 Performance evaluation . . . 108

9 Performance Results **111**
9.1 Method 1 . . . 111
9.1.1 Block sizes . . . 111
9.1.2 Performance test . . . 113
9.2 Method 2 . . . 118
9.2.1 Performance test . . . 118
9.3 Merged Methods . . . 121
9.4 Brief study of performance with single-precision . . . 122

10 Conclusion **125**

11 Further Research **127**

Appendix A Nomenclature **129**

Appendix B Platforms Specification **131**

Appendix C Project Description Provided by DHI **133**

Appendix D Bandwidth Test **135**

Appendix E Source Code to Performance Test **137**

Bibliography **143**

List of Figures

1.1 Illustrations of GPU Computing [23]. 2
1.2 Flowchart over workflow for scientific computing from idealization through discretization to simulation. 3

2.1 Amdahl's Law for parallel fraction f of a program. 10

3.1 Flowchart for GPGPU programming, [44]. 13
3.2 Illustration of relation between threads, blocks and grids [29, fig. 2-1]. 14
3.3 Overview of the Fermi GF110 architecture [2]. 16
3.4 Screenshot of one of the implemented kernels run in NVIDIA Visual Profiler. 22

4.1 Overview of the bathymetry, water surface elevation and depth. . 26
4.2 Staggered grid in (x,y)-space, [21, fig. 3.1]. 28
4.3 Time centering overview of computation cycle, [21, fig. 6.1]. . . . 29
4.4 Grid notation for the convective term in the x-momentum equation, [21, fig. 4.5]. 32
4.5 Grid notation for the cross term in the x-momentum equation, [21, fig. 4.6]. 33

5.1 Workflow of PCR algorithm in the 8-unknown case, [47, fig. 2]. Equations are labelled $e1$-$e8$. Equations in a yellow rectangle form a independent system. Arrows are omitted in step 2 for clarity. 44
5.2 Workflow of CR algorithm in the 8-unknown case, [47, fig. 1]. Equations are labelled $e1$-$e8$. 46
5.3 Workflow of CR-PCR hybrid algorithm in the 8-unknown case, [47, fig. 4]. Equations are labelled $e1$-$e8$. Details about PCR are omitted see Figure 5.1. 47

6.1 Explansion of *a system* in the grid. 49
6.2 Workflow of the C application. 51
6.3 Profiling using AQtime of MIKE 21 HD on a 512×512 grid with 1000 time steps performed by DHI. Specification of the test system is given in Appendix B Table B.3 page 132. 58
6.4 Profiling using AQtime Standard vers. 7 of the C implementation on a 512×512 grid with 1000 time steps. Specification of the test system is given in Appendix B Table B.3 page 132. 59
6.5 Convergence test. The truncation error follow 2nd order of convergent. 61
6.6 Illustration of the initial wave and how it has spread after 10 time steps. 63
6.7 Illustration of the wave after 20 and 30 time steps respectively. . 63
6.8 Illustration of the wave after 40 and 50 time steps respectively. . 63
6.9 Illustration of the wave after 60 and 70 time steps respectively. . 63

8.1 Execution time overview for method S1N. 73
8.2 Access pattern for S1N for a x- and y-sweep. 75
8.3 Flowchart through one simulation time step in the naive S1 approach shown in gray and the modified x-sweep in blue. 76
8.4 Execution time overview for S1N and S1N2. 76
8.5 Needed values for calculation of a mass and momentum equation at (j,k) for respectively $\zeta^{n+1/2}$, $q^{n-1/2}$ p^n/p^{n+1} in an y-sweep. . . 78
8.6 Missaligned access pattern where the first thread accesses values in a separate 128 byte segment. 78
8.7 Perfectly coalesced access pattern 79
8.8 Missaligned access pattern where the last thread accesses values in a separate 128 byte segment. 79
8.9 Impact on occupancy when varying block size 8.9a and registers per thread 8.9b for S13. 90
8.10 Execution time overview for method S2N. 96
8.11 Access pattern for S2N for an x- and y-sweep. 97
8.12 Flowchart through one simulation time step in the naive S2N and modified S22 approach. 98
8.13 Execution time overview for S2N and S22. 99
8.14 Impact on occupancy when varying block size 8.14a and registers per thread 8.14b for S24. 104
8.15 Effective bandwidth for all the different parallel solvers. 109
8.16 Effective bandwidth for all the different parallel solvers. 110

9.1 The execution time for different block and system sizes. 112
9.2 Execution time for three chosen systems sizes 2048×2048, 2560×2560 and 3008×3008 using approach S1 and the C implementation. . . 113
9.3 Speedup of all S2 implementations. 114

9.4 Effective bandwidth of all S2 implementations. 115
9.5 Scaling of the execution time for the CPU and S14 in log-log plot. 116
9.6 Speedup of all S2 implementations. 118
9.7 Effective bandwidth of all S2 implementations. 119
9.8 Effective bandwidth for xBuild24. 120
9.9 Speedup for S14 and S24 against the CPU. 121
9.10 Speedup of S14 and S24 in single- and double-precision against the CPU. 122
9.11 Effective bandwidth of S14 and S24 in single- and double-precision. 123

List of Tables

3.1 Overview of device memory. Note that all outlined memory types can be read from and written to. 16

5.1 Algorithmic operations and algorithmic steps for the different algorithms where n is the system size, m is the intermidiate system size and k represænts the step. Both n and m is assumed to be a power of 2. 46

6.1 Execution time of MIKE 21 HD and the C implementation for different system sizes over 100 time steps. Specification of the test system is given in Appendix B Table B.3 page 132. 60

7.1 Profiling the naive application `S1` in single-precision to calculate the instruction to byte ratio. 68
7.2 Measurement the ratio of peak performance between single- and double-precision on GeForce GTX 590 and Tesla C2070. 69

8.1 Runtime for the combined data transfer for host-device and devise-host for respectively pageable and pinned memory allocation. . . 72
8.2 Metrics and events from NVIDIA Visual Profiler for the naive approach `S1N`. 74
8.3 Gather performance metric for one transposing of one array with size 2048 × 2048. 77
8.4 Comparing execution time of the seqential C implementation on the CPU against `S1N` and `S1N2` on a 2048×2048 system for 100 time step in double-precision. 77
8.5 Metrics/Events from NVIDIA Visual Profiler for `S12`. 80
8.6 Metrics from NVIDIA Visual Profiler for `S12` when caching and non-caching in L1. 81
8.7 Metrics from NVIDIA Visual Profiler for `S12` with 48 kB L1 cache size. 82

8.8 Comparing execution time of the CPU against S12 on a 2048×2048 system for 100 time step in double-precision. 82
8.9 Events from NVIDIA Visual Profiler for S12. 83
8.10 Events from NVIDIA Visual Profiler for S12 and S13. 88
8.11 Comparing execution time of the CPU against S13 on a 2048×2048 system for 100 time step. 88
8.12 Profile counters for yBuild in S13. 91
8.13 Comparing execution time of the CPU against S13 and S14 on a 2048×2048 system for 100 time step in double-precision. 93
8.14 Metrics and events from NVIDIA Visual Profiler for the naive approach S2N. 95
8.15 Metrics and events from NVIDIA Visual Profiler for the S22 approach. 98
8.16 Comparing execution time of the CPU against S2N and S22 on a 256 × 256 grid over 100 time steps in double-precision. 99
8.17 Memory metrics and events from NVIDIA Visual Profiler for the S23 approach. 100
8.18 Comparing execution time of the CPU against S24 on a 256 × 256 grid over 100 time steps in double-precision. 101
8.19 Different approaches to achieve better global load efficiency for S24. 102
8.20 Memory metrics and events from NVIDIA Visual Profiler for the S24 approach. 102
8.21 Instructions metrics and events from NVIDIA Visual Profiler for the S23 and S24 approach. 103
8.22 Comparing execution time of the CPU against S24 on a 256 × 256 grid over 100 time steps in double-precision. 103
8.23 Memory metrics and events from NVIDIA Visual Profiler for the different PCR solver optimizations. 106
8.24 Memory metrics and events from NVIDIA Visual Profiler for the different CR-PCR solver optimizations. 108
8.25 Comparing execution time of the CPU against S24PCR with shared memory and S24CR−PCR with shared memory on a 256×256 grid over 100 time steps. 108

9.1 Profile counters for xBuild in S24. 119

A.1 Nomenclature 130

B.1 Specifications of test environment. 131
B.2 Specifications of GPU NVIDIA GeForce GTX 590 used in tests. . 132
B.3 Specifications of DHI test environment. 132

Chapter 1

Introduction

In today's world the focus on environment and climate is higher than ever. Especially the changes in rivers and coastal areas have crucial impact on peoples lives all over the world [11]. The world saw in 2004 the devastating effect of a tsunami killing over 230,000 people in fourteen countries bordering the Indian Ocean. The tsunami had waves up to 30 meters high and the world donated more than $14 billion U.S. dollars in humanitarian aid [43]. Having efficient tools to try and predict such events has potential to save hundreds of thousands of lives and billions of dollars. It is therefore essential in today's modern society.

DHI have developed a 2D free-surface flow numerical engine called MIKE 21 HD (hydrodynamic module) in order to simulate water movements [9]. MIKE 21 HD was first developed back in the 1980s and has since then gone though a lot of improvements regarding precision, ability and simulations speed to combine different variations, according to DHI. The application can simulate water movement in lakes, estuaries, bays, coastal areas and seas, based on rain, tidal variation, wind etc, but also including prediction of tidal hydraulics, wind and wave generated currents, storm surges, waves in harbours, dam-breaks and tsunamis (see project description provided by DHI in Appendix C page 133).

MIKE 21 HD uses a set of hyperbolic partial differential equations called shallow water equations, which describe the flow below a pressure surface in a fluid. These equations are solved numerically on a uniform rectangular grid, using a finite difference method (FDM). The solution scheme is the Alternating Direction Implicit (ADI) method. The ADI method split the finite difference equations into two stages per time step, treating one operator implicitly at each stage, for which the equations are solved in both of the stages. The system of equations introduce a tri-diagonal coefficient matrix, which can be solved efficiently.

The simulation speed of the program is very important, because it determines how big and how many problems can be solved in a given amount of time. Sometimes in order to get accurate understanding of the changes in a given area, hundreds of simulations have to be run. Therefore, improving the execution speed has the potential to increase type and size of optimization problems, where MIKE 21 HD is applicable and thereby open new market segments for DHI. For this reason the focus of the project will be on improving the simulation speed, while maintaining the accuracy of the program. This will be performed by implementing the program to run on a graphics processing unit (GPU) to exploit the massively parallelism of the architecture. Using the GPU to perform scientific calculations can be beneficial to problems (such as this), where the calculations are independent of each other and therefore can be performed in parallel. The technology for doing this is relatively new. In 2006 NVIDIA published CUDA to run on NVIDIA CUDA-enabled GPUs as the world's first solution for general-computing on GPUs [22].

In this project will the parallel programs be implemented in CUDA C and thereby only be executable on CUDA-enabled GPUs. Further will the programs be optimized to GPUs based on the *Fermi* architecture, which is optimized for scientific applications and was the latest CUDA architecture, when the project started. On March 23 2012 was the Kepler architecture launched, but unfortunately we did not have the chance to test on this architecture.

The strength of GPGPUs lies in the many-core architecture, wide SIMD parallelism[1] and scalability.

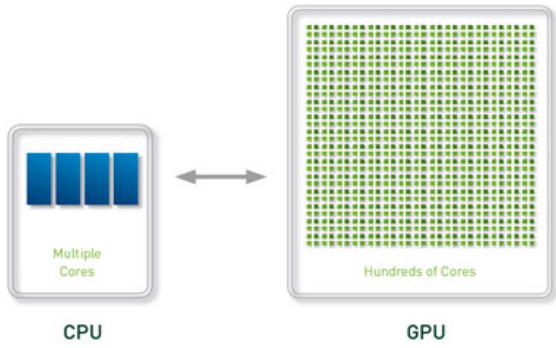


Figure 1.1: Illustrations of GPU Computing [23].

The idea of GPU computing is to utilize the available high-performance resources by using the central processing unit (CPU) and GPU together in a

[1]NVIDIA call it Single Instruction, Multiple Threads (SIMT), but it is essentially Single Instruction, Multiple Data (SIMD).

heterogeneous co-processing computing model. That is to use the GPU as an accelerator for an application. The sequential part of the application is runs on the CPU, while the parallel computationally-intensive part is accelerated by the GPU [23]. Consequently, the available resources will be exploited in a manner of what their force are. The CPU in general has a higher clock frequency, but can only perform a couple of operations simultaneously, which is good for serial computations. The GPU have a lower clock frequency, but can perform thousands of tasks at the same time, which is good for parallel computationally-intensive computations (massive parallel). This is exactly how we intend to utilize the GPU in this project to speedup the heavy independent computations in MIKE 21 HD and throughout improve the simulation speed.

1.1 Validation of Scientific Computing

Using computers to simulate the real world is the main task in scientific computing. Figure 1.2 illustrate the workflow from having a scientific problem to developing a mathematical model which describes the phenomenon. For then to discretize to a numerical formulation, which can be solved on a computer.

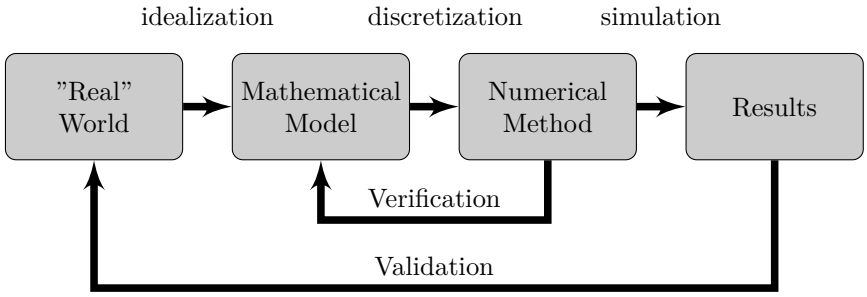


Figure 1.2: Flowchart over workflow for scientific computing from idealization through discretization to simulation.

To insure that the discretization of the mathematical model behave as expected and desired should a verification of the numerical method be performed. Further, to insure that the numerical model simulates the "real" world problem, as desired, should the obtained solution be validated against the scientific problem.

In this project will the obtained solutions frequently be compared to MIKE 21 HD to make sure the programs always obtain the same results. MIKE 21 HD has gone through a lot of development since the 1980s. The aim of this project is therefore not to change or improve on the numerical formulation or correctness of MIKE 21 HD, but on the request of DHI to develop a parallel solution scheme

that obtain the same solution as MIKE 21 HD. Thereby is our obtained results validated by comparison to the result returned by MIKE 21 HD. Since we need to obtain the same solution as MIKE 21 HD will the mathematical model and numerical discretization used in MIKE 21 HD also be used in this project. However, we have discretizated the mathematical model and formulated the problem that shall be solved ourself, analogue to the discretization by DHI, to insure correctness and better understanding. A verification will be applied just to verified that the truncation error embedded by the approximations behave as expected. Consequently, since the primary task of the project is to develop a fast parallel solution scheme will focus throughout the report be on this, while maintaining the same solution as MIKE 21 HD.

1.2 Proposal from DHI

The original project description provided by DHI can be found in Appendix C page 133.

The project is proposed by DHI in order to investigate the potential of moving the existent MIKE 21 HD implementation form the CPU to the GPU. Consequently, to research whether it is beneficial to formulate a parallel solution scheme of the current numerical algorithm used in MIKE 21 HD or if other algorithms, there better utilize the resources on a GPU, should be used.

DHI present following approach for the project; first the central subroutines of MIKE 21 HD must be implemented in a sequential C version. Second shall a parallel solution scheme, based on the sequential C version, be implemented in CUDA C. All performed calculations and all used data types shall be in double-precision. The sequential version serves the purpose of having a correct base code, which always produces the same result as MIKE 21 HD, and thereby to verify the correctness of the different GPU implementations. Additionally to compare performance of the CPU and the GPU implementations. The performance comparison is performed on the C implementation, since the MIKE 21 HD FORTRAN implementation is more complex and therefore does not provide a fair comparison. However, a profiling of the MIKE 21 HD FORTRAN implementation will be conducted to make sure that it has the same bottlenecks as the C implementation. Furthermore is it important that the developed programs obtain the same result as MIKE 21 HD. This means that the validation of the implemented applications will be obtained through comparison to MIKE 21 HD.

DHI have also states in the project description, that a drastic improvement in simulation speed has the potential to change the type of optimization problems, where MIKE 21 HD is applicable, and thereby open new market segments for DHI. The project is therefore highly relevant.

1.3 Thesis Statement

This project has the purpose to examine and improve the simulation speed of the central algorithm of the hydrodynamic model in MIKE 21 Flow Model by DHI. The focus will be on utilizing that the algorithm can be performed in parallel and thereby increase the simulation speed by solving it on a GPU.

The primary objective of the project is:

- **How can shallow water fluid flow equations be solved efficiently in parallel using a GPU and how much can this improve the performance of MIKE 21 HD?**

In order to answer this question it is necessary to answer the following:

- How can the existing MIKE 21 HD FORTRAN code be converted into a sequential C program?
- How can this algorithm efficiently be solved in parallel using CUDA C?
- How can we find an improved strategy to solve the problem by utilizing the GPU architecture?
- Are there other numerical algorithms, which are more beneficial in order to utilize the GPU architecture when solving the problem?

The result of the project will be C and CUDA C programs which correspond to the core structure of MIKE 21 HD FORTRAN code. Throughout the project will profiling of the implementations be used to deduce performance studies in order to locate bottlenecks and document change in performance.

The focus on the sequential C implementation is mainly on the correctness of the program compared to MIKE 21 HD, so the obtained results from the parallel solutions can be validated against the serial. It is also used for comparison of the simulations speed.

1.4 Scope of the Project

The subject of this report is far more extensive than the scope of this project. The areas that the report will focuses on is therefore confined and will not be discussed further. Listed below are the areas that fall outside the scope of this report.

MIKE 21 HD. The implemented programs will not correspond to the full MIKE 21 HD, since this will be too extensive a task. However, the most important parts and the most intensive computations in the MIKE 21 HD will be implemented, why the outcome in this project will be realistic and useful in manner of a fully CUDA C implementation of MIKE 21 HD. The CUDA C application will only be implemented to handle a set up with coast along the boundary and water in all inner points.

Parallel programming. The parallel solutions schemes will only be implemented and optimized to NVIDIA Fermi architecture using NVIDIA parallel computing architecture CUDA (Compute Unified Device Architecture). The focus will especially be on the NVIDIA GeForce GTX 590[2], since this card nearly have same specifications as NVIDIA GeForce GTX 580, which DHI are in possession of. Thereby will the obtained results be comparable an feasible for them too. In fact, since only one of the GF110 chips on the GTX 590 are used will DHI be able to obtain even better results using the GTX 580.

Notice that the implementations should also be able to be executed on newer models of NVIDIA CUDA-enabled GPUs.

OpenCL. The parallel solutions schemes will not be implemented in OpenCL, by which there will not be performance studies of the difference between using OpenCL or CUDA. This implies that the differences with hardware platforms such as ATI vs. NVIDIA not will be investigated.

Optimization on CPU. The goal of the project is mainly the parallel implementation of MIKE 21 HD. Therefore the sequential implementation will be performed reasonably, but the main optimization will be on the parallel program.

Applied optimization techniques will be how to utilizing the capacity of NVIDIA CUDA-enabled GPU hardware in a cleaver way, but also to investigate whether there are other numerical algorithms that can solve the shallow water equations and perform better parallelism than the used solution scheme in MIKE 21 HD.

[2]Test environment specifications is available in Appendix B page 131

1.5 Learning Objectives

The learning objectives for the project are listed in the following

- In qualified manner formulate, analyze and solve problems within a limit project.
- Solve a relevant engineering problem, where acquired knowledge and skills are used.
- Evaluate and summarize results, and account for the results in a logical and structured technical report.
- Apply principles for numerical discretization and approximation.
- Implement and verify numerical algorithms for solving partial differential equations, in particular solving the shallow water fluid flow equations (hyperbolic PDE).
- Make a performance study that documents the obtained experiences, and clarifies changes in the simulation speed.
- Apply obtained theoretical knowledge and experiences to optimize a "real world" engineering problem.
- Investigate opportunities for utilizing the massive parallel computations that the GPU architecture provide.
- Skillfully analyze, design and implement parallel programs for Scientific Computing problems and use modern architectures and tools to achieve efficient programs. In particular a parallel solution scheme for MIKE 21 HD using a GPU (graphics card).

We will through the project obtain these competencies.

1.6 Thesis Structure

In this thesis will the needed theory first be described. It will then be used to solve the problems throughout the thesis and at last will the results be shown and discussed. The systems on which the tests are performed can be seen in Appendix B Table B.1.

Here is an overview of what the individual chapters will contain.

Chapter 2 Investigates the potential performance gain by developing a parallel solution scheme.

Chapter 3 Describes parallel programming in CUDA, how to optimize applications and the functionality of the Fermi architecture.

Chapter 4 Derives a discretization of the shallow water equations and formulates the systems that needs to be solved.

Chapter 5 Describe the different tri-diagonal solution algorithms, which will be used in the project.

Chapter 6 Describe the functionality of MIKE 21 HD and the sequential C implementation and compare the performance of this two.

Chapter 7 Gives an overview of the parallel CUDA C implementations.

Chapter 8 Goes through the optimization steps of the parallel CUDA C implementations.

Chapter 9 Investigate and discusses the results obtained throughout the project.

Chapter 10 Summing up the thesis and makes a conclusion of the work and results.

Chapter 11 Describes what aspects would be interesting to analyse and investigate further.

Appendix Contain nomenclature used in the thesis, the specifications for the different test systems used in the project, key values about the Fermi architecture and the original project description provided by DHI.

Chapter 2

Scalability and Expectations of Speedup

What benefit can we expected by parallelizing the sequential MIKE 21 HD application? Before impelemting a given problem on a GPU is it recommend to have some prior knowledge about how well the problem can be parallelized, since tasks that cannot be sufficiently parallelized never should be executed on a GPU. Therefore, to state expectations for speedup from a parallel CUDA implementations are there commonly distinguished between scalability with

Strong scaling Theoretical upper bound of decreasing the solution time as more processors are used for a fixed problem size (speed-up).

Weak scaling Theoretical upper bound of keeping the solution time constant as problem size increases by adding more processors with a fixed problem size (scale-up).

This chapter is based on knowledge which can be obtained by [32, sec. 2.1.3].

2.1 Strong Scaling

Normally, strong scaling is equated with Amdahl's law. Amdahl's law state the expected speedup by parallelizing a fraction of a sequential program. Assume the program has a parallel fraction f. This implies that the execution time can be split into

$$T(1) = f \cdot T(1) + (1 - f) \cdot T(1)$$

and with p processors available

$$T(p) = \frac{f}{p} \cdot T(1) + (1 - f) \cdot T(1)$$

We then have Amdahl's Law

$$S(p) = \frac{T(1)}{T(p)} = \frac{1}{(1 - f) + \frac{f}{p}} \tag{2.1.1}$$

where $S(p)$ is the maximum expected speedup. We see from Amdahl's Law (2.1.1) that the larger the parallelizable fraction f is, i.e., f is close to 1, the greater speedup can be expected. However, increasing the number of processors p does also have an impact on the performance. Figure 2.1 illustrate Amdahl's Law for different parallel fraction f of a code

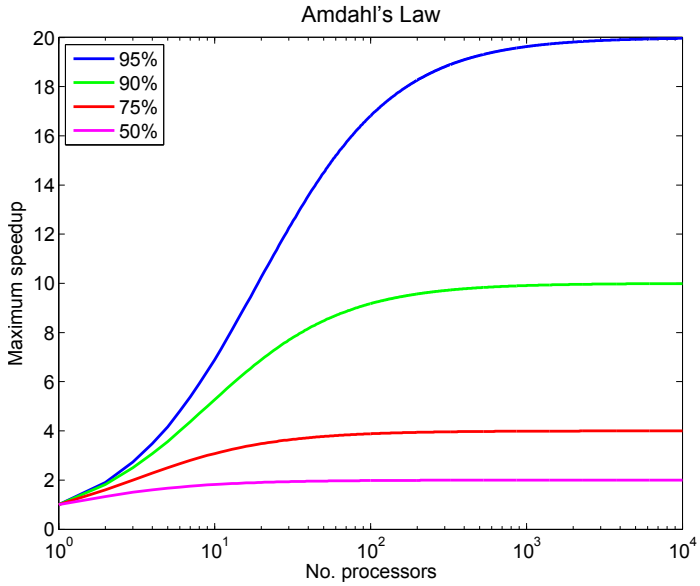


Figure 2.1: Amdahl's Law for parallel fraction f of a program.

This implies that parallelizing that part of the code, where the majority of the time is spent, is very important to gain a beneficial performance of the GPGPU. For instance, if a large number of processors are available and the sequential fraction is 20%, the maximum speedup is only 5.

2.2 Weak Scaling

Usually, weak scaling is equated with Gustafson's Law. Gustafson's Law state how much the problem size can increases by adding more processors with fixed work, keeping constant execution time. Thus the maximum speedup of a program is

$$S(p) = p + (1 - f)(1 - p) \tag{2.2.1}$$

where p again is the number of processors and f the parallel fraction. Note, it is assumed that the parallel fraction remains constant.

2.3 Applying Strong and Weak Scaling

Having some kind of understanding on how the application will scale can give an expectation of attained speedup when parallelizing a sequential application.

To enable the opportunity for DHI solve larger problems and/or increase accuracy will demand the possibility to increase the problem size and thereby fill the available processors. This exhibit that the application has weak scaling qualities. Thus applying Gustafson's Law to determine an upper bound for speedup. Additional, one could conceive that DHI also would be interested in knowing how the application will scale if one execute the same problem on different hardware, thus apply Amdahl's Law could determine an upper bound for the speedup.

In the grid are each row or column independent on the others. So, for instance, if each thread calculate one row/column in the grid and that the system is big enough such as there are sufficiently work to do, one can say that the parallel fraction are one. Consequently, this will, cf. Gustafson's Law, result in a parallel application, which will scale linear. Thus one doubling the problem size will double the speedup. However, one should note that additional overhead due by parallelization and speedup caused by cache effects are not taken into account. The same applies to memory bandwidth limitation and sufficiently work and processors available.

These theoretical laws verify two fundamental conditions for parallel computing, since they roughly can be summarize to

- The extent of the parallelism in the application need to be sufficiently to utilize the available resources.
- The problem need to be sufficiently large, so more processors can be utilized.

Chapter 3

CUDA Theory

The motivation for using the GPU to perform scientific computing lies in the power of the massively parallel architecture. While a modern CPU commonly has 4-16 cores the GPU have hundreds. Even though the cores of a CPU often is individually faster the volume and the intelligent memory architecture means that there are huge processing power available in the GPU to solve parallel problems.

In this chapter will the CUDA and Fermi architecture be explained. Further more will different optimization techniques that are beneficial CUDA programs be described. The chapter will be based on knowledge which can be obtained by [29], [32], [33] and [12], where also more detailed descriptions can be found.

3.1 Processing Flow for GPGPU

It is important to be aware of the processing flow when programming GPGPU for scientific computing. The flow is illustrated in Figure 3.1.

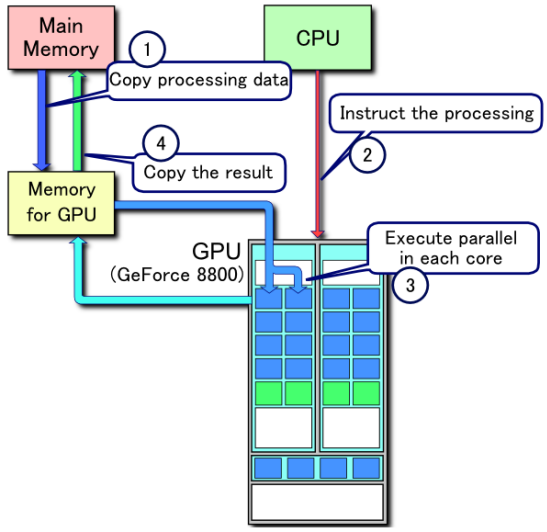


Figure 3.1: Flowchart for GPGPU programming, [44].

The flow can be divided into following steps

1 Copy data from host (CPU memory) to device (GPU memory).
2 Send instructions from host to device.
3 Execute calculations in parallel on the GPU.
4 Copy result from device to host.

Especially step 1 and 4 can be very slow because of the low bandwidth between host and device. For this reason data transfers between host and device should be minimized. In fact, there are three general suggestion when creating high-performance GPGPU programs [12, chap. 1]

- Get the data on the GPGPU and keep it here.
- Give the GPGPU enough work to do.
- Focus on data reuse within the GPGPU to avoid memory bandwidth limitations.

3.2 CUDA C

CUDA (Compute Unified Device Architecture) is the architecture that enables NVIDIA GPUs to execute parallel programs. CUDA C works as an extension of C, where additional functions are added, e.g., functions to allocate arrays on the GPU `cudaMalloc()` and copy data from the CPU to the GPU `cudaMemcpy()`.

Parallel CUDA applications are called kernels, which are executed in the host code. A kernel describes the work that a given thread shall perform based on a uniquely assigned thread index organizes in thread blocks and grids of thread blocks. In order to control the parallel executions are a virtual layout with three layers used; threads, blocks and grids. A thread is the core element that performs the calculations in parallel. Threads are divided into groups called blocks (or thread blocks) which again are divided into grids, see Figure 3.2 for the relation between threads, blocks and grids.

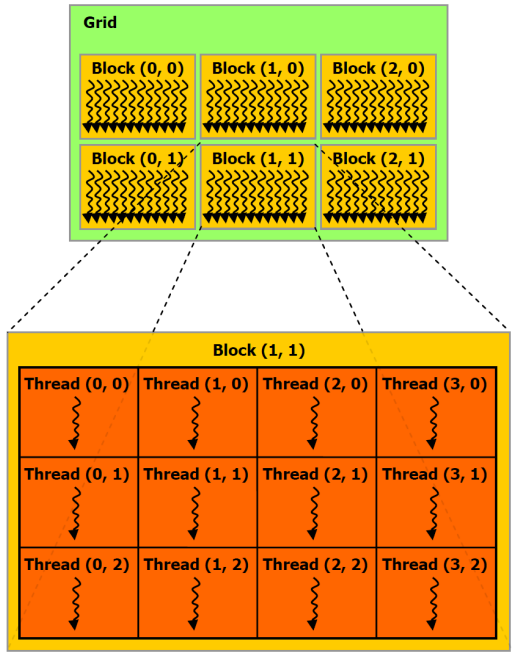


Figure 3.2: Illustration of relation between threads, blocks and grids [29, fig. 2-1].

All threads within a block is processed in parallel on the same multiprocessor and all has access to the same shared memory, see Section 3.6. This allows threads to cooperate within the block and efficiently share data with each other. Since blocks are distributed to different multiprocessors can they on the other hand not easily cooperate. For this reason is it important that blocks can be executed

independently. Additionally, one should keep in mind that even though threads can be viewed as being performed in parallel they are physically calculated in groups of 32 in so called warps. For this reason should the size of the block be a multiple of 32 and threads within a warp should have the same code path and access memory addresses close together.

The size and dimension of the grids and blocks are decided by the programmer with some restrictions, see Appendix B.2. This allows for a given application to be optimized based on block and grid sizes and dimensions.

3.3 Fermi Architecture

In order to develop efficient parallel GPGPU applications is it important to have knowledge about the hardware and architecture it is executed on. This will give an indication and comprehension on how the application will gain performance in parallel on the GPU. Therefore, a brief description will be given of the architecture.

CUDA computing graphic architecture *Fermi* is the second line of architectures developed by NVIDIA. In this project will code be optimized for and tested on the NVIDIA GeForce GTX 590 (2x GF110 chip), with have Compute Capability 2.0. The graphic card consists of 16 streaming multiprocessors with 32 CUDA cores each with gives a total of 512 parallel processing cores. Each CUDA processor can perform integer arithmetic logic and simple floating point operations such as addition, subtraction, multiplication. The architecture use now the full IEEE 754-2008 32-bit and 64-bit floating-point standard, so for instance the multiply-add instruction (FMA) can be performed without losing precision in the addition. Further are there four special function units per multiprocessor (SFU), which executes special instructions such as sin, reciprocal and square roots. See Figure 3.3 for an overview of the architecture.

Figure 3.3: Overview of the Fermi GF110 architecture [2].

The hardware also consists of different memory types located both on- and off-chip as illustrated in Table 3.1.

Memory	Location	Scope	Lifetime	Latency
Register	On-chip	Thread	Thread	1 cycle
Shared Memory	On-chip	Block	Block	2-4 cycles
Global Memory	Off-chip	Global	Application	400-800 cycles

Table 3.1: Overview of device memory. Note that all outlined memory types can be read from and written to.

Further description of the different memory types that are available on the hardware see Section 3.5-3.7.

3.4 Data Transfer Between Host and Device

The peak theoretical bandwidth between host and device memory is very slow, especially compared to the peak theoretical bandwidth internal on the device. The graphic card is connected through a CPI-E 2.0 x16 bus speed. In NVIDIA CUDA C/C++ SDK Code Samples are there `bandwidthTest`, which test the bandwidth internal, host to device and device to host. The test result for the GTX 590 see Appendix D on page 135.

To achieve higher bandwidth between the host and the device one could using pinned memory instead of the commonly pageable memory. CUDA C Runtime API provides functions to pinned host memory. This has several benefits, e.g., bandwidth between host and device memory is higher and is used when data transfer between host and device is performed concurrently with kernel execution. Since the device can access the memory directly, it can read and writhe with higher bandwidth. So instead of using `malloc()` there allocate pageable host memory can one use `cudaHostAlloc()` to allocate and `cudaFreeHost()` to free pinned host memory.

One should keep in mind that excessive use of pinned memory can have a negative impact on the system performance, because of the less available memory to the system.

3.5 Global Memory

Global memory is a read/write off-chip DRAM, available to all threads in the grid. Global memory access has a low memory throughput (163.87 GB/s for GTX 590) compared to faster on-chip memory and should be minimized. This can be done using the faster on-chip memory rather than slower global memory, see Section 3.6.

Accessing global memory are cached in either L1 and/or L2, which can be configured in compile time. Using the compiler flag `-Xptax -dlcm=ca` will result in caching both in L1 and L2, compiler flag `-Xptax -dlcm=cg` will lead to L2 caching only. If both the L1 and L2 cache is activated then global memory requests are serviced with 128 byte memory transactions. If only L2 cache is activated then memory requests can be serviced with 32 byte memory transactions. Thus only caching in L2 can reduce over-fetch in case of scattered memory access.

For most applications will some global memory access be necessary. It is therefore important that global memory accesses are done as efficiently as possible to minimize wasted bandwidth. Hence, all global memory accesses should be performed coalesced.

3.5.1 Coalesced memory access

Since the global memory requests are serviced with 128 byte memory transactions would it obviously be most efficient if all those 128 byte was needed by the threads. Therefore, in order to achieve efficient global loads, the accesses should be coalesced such that the threads in a warp request values in global memory that are located next to each other and falls into as few 128 byte transactions as possible.

If a single thread requests an element from a different segment than all the other threads, a whole 128 byte segment must be transferred to answer that

single thread request. Thereby will bandwidth be wasted. So in order to achieve high efficiency and few memory transfers the accesses must be coalesced, which implies threads request aligned data which fall into full 128 byte segments. In order to obtain this is it very important that the threads access correctly in the memory layout which is row-wise when programming in C.

Alternatively the global load efficiency can in some cases be improved by turning off the L1 cache, since smaller memory transactions then can be performed. This will improve global load efficiency if the access pattern is scattered, since less unnecessary data has to be transferred. However, this means that the fast and often very useful L1 cache can not be used, which can result in longer execution times even though a higher global load efficiency is achieved.

3.6 L1 Cache and Shared Memory

The same 64 kB on-chip memory on each multiprocessor is used for L1 cache and shared memory. All threads in the same thread block has access to the same elements in shared memory. The size of these two is either that L1 has 16 kB and shared memory 48 kB or vice versa. This can be configured from the host with the CUDA API routine `cudaFuncSetCacheConfig(myKernel, cacheConfig)` where `myKernel` is the name of the kernel and the `cacheConfig` options are

- cudaFuncCachePreferShared: shared memory is 48 kB and L1 16 kB
- cudaFuncCachePreferL1: shared memory is 16 kB and l1 48 kB
- cudaFuncCachePreferNone: no preference

While caching in L1 happens automatic can shared memory be viewed as a user-managed cache. Shared memory is especially beneficial when data has to be used and/or modified several times, while L1 cache is great at reducing traffic to global memory, smooth out some misaligned, scattered access patterns and helps with registers spilling.

Shared memory has 32 memory banks, where successive 32-bit words are assigned to successive banks. Each bank has a bandwidth of 32 bits per two clock cycles. When using shared memory one should be aware of bank conflicts.

3.6.1 Shared memory bank conflicts

A bank conflict occurs when two or more treads in a warp accesses different 32-bit words in the same bank. When bank conflicts occurs shared memory accesses is serialized and the throughput is decreased by a factor equal to the number of separate memory requests. For 64-bit access a bank conflict only occurs if there is a bank conflict in either of the half warps. This is an improvement compared to Compute Capability 1.x where 64-bit shared memory accesses normally would occur with a two-way bank conflict.

3.7 Registers

Registers are on-chip automatic memory and are the fastest memory on the GPU. There are 32,768 32-bit registers available per multiprocessor on devices with Compute Capability 2.x. Registers are private to each thread and data cannot be read by other threads.

A thread is limited to a maximum of 63 registers per thread, but this limit can be decreased for all kernels at compile time with the compile option `–maxregcount=N`, or for a given kernel with the `__launch_bounds__()` qualifier in the definition of a `__global__` function. However, if threads need more registers that are available can this result in register spilling.

3.7.1 Register spilling

If a thread needs more memory to store automatic variables than the available registers, then register spilling will occur. This means that the threads will use local memory to store the excess data. This can be very inefficient, since the local memory space resides in device memory. Thus register spilling can have an impact on the performance by increasing memory traffic or instruction count.

Register spilling might not always be a problem, since it might be partly or entirely contained in the L1 cache and if the application is not compute/instruction bounded, this does not so much. Also it might account for an insignificant amount of global memory transfers. It should therefore always be investigated how significant the spilling are and based on that act correctly; if register spilling is problematic one could try increasing the limit of registers per thread, non-caching loads for global memory and/or increase L1 cache size to 48 kB.

3.8 Latency Hiding

The number of clock cycles it takes for a warp to be ready to execute its next instructions is called latency. In order to fully utilize the hardware should the multiprocessor perform work every clock cycle. This implies that all latency should be ”hidden” with instructions from other warps that are ready to execute. Obviously, longer latencies are more difficult to hide and the more warps that are available on the multiprocessor the easier it is to hide latency. This motivates further to use fast memories, where latency will be lower and to achieve high occupancy see section 3.9.

The way to hide latency is, however, not always to run more threads per multiprocessor or to have more threads per thread block (Thread Level Parallelism (TLP)). One could try hid arithmetic and/or memory latency using fewer threads (Instruction Level Parallelism (ILP)).

Arithmetic latency. Generally we know that accessing registers costs no extra cycles. However, read-after-write latency is ≈ 24 cycles and 400-800 cycles for memory. A example is given in Listing 3.1

Listing 3.1: Example of hiding register dependencies.

```
1 x = a + b;  // take approx 24 cycles to execute
2 z = x + d;  // dependent, must wait for completion
3 y = a + c;  // independent
```

We see that line 2 is dependent and can first be executed when line 1 is finish. However, line 3 is independent and can start anytime. Therefore, one could maybe obtain better performance by interchange line 2 and 3, such that the latency for line 1 can be hidden by doing other operation in the meantime.

Overall, it is usually recommended to apply more threads to supply the needed parallelism (TLP). Additional, one can use parallelism among the instruction for each single thread (ILP).

3.9 Occupancy

Occupancy is the ratio of the number of resident warps per multiprocessor to the number of theoretically possible. For devices of Compute Capability 2.x are the maximum number of resident warps per multiprocessor 48. Occupancy is therefore a measure for how well the multiprocessor can hide latencies and keeping the hardware busy, see Section 3.8. Thus one can calculate the occupancy by

$$\text{Occupancy} = \frac{\text{Resident blocks per SM} \cdot \text{Threads per block}}{\text{Maximum threads per SM}} \tag{3.9.1}$$

The occupancy can be limited by a number of factors like block size, registers used per thread, shared memory. The limits on all of these factors is depended of Compute Capability and can be seen in Appendix tab:cudaspecs. The occupancy and the limiters can easily be identified using CUDA Occupancy Calculator, which is a tool in CUDA Toolkit provided by NVIDIA [27].

It should be mentioned that high occupancy does not necessarily equal high performance, but low occupancy will properly equal low performance.

3.10 `nvcc` Compiler

The source code can be a mix of host and device code, thus NVIDIA compiler `nvcc` separate the `.cu` code, such that host code is compiled on the available C/C++ compiler on the host platform and the device code is compiled by NVIDIA assembler or binary instruction.

The NVIDIA CUDA Driver API is a low-level C API, provide to link to options as `-ptx` and `-cubin`. Linking with these will control specific phases of the compilation. `-ptx` is intermediate assembler code for NVIDIA GPU standing for Parallel Thread eXecution. `-cubin` is a CUDA binary. These linker options can be useful when investigating what really happens behind the scene and thereby make it possible to optimize the application very specific and determined [7].

`nvcc` provide some useful compiler options, like

- `-arch=sm_20` define Compute Capability to 2.0, insuring double precision.
- `-maxrregcount=N` specifies the maximum number of registers per thread.
- `--ptxas-options=-v` or `-Xptxas=-v` return register, shared and constant memory usage per kernel.

`nvcc` supports restricted pointers via the keyword `__restrict__`. This keyword is used to tell the compiler that the pointers do not overlap. Notice that it is your own responsibility that this never will be violated. Thus the compiler can optimize the code by reordering and do common sub-expression elimination at will. Consequently, this can reduce memory accesses and number of computations. However, this can increase the register pressure, thus it can have a negative impact on the performance.

3.11 NVIDIA Visual Profiler

The NVIDIA Visual Profiler is a very useful tool to evaluate an optimize implemented kernels. It can measure many different events and calculate metrics as runtime, warp divergence, shared memory bank conflicts etc.

For this reason it is very efficient tool. Thus one can locate bottlenecks an problems in a kernel. Use the different provided events and metrics to investigate how well a given kernel performance compared to the hardware, but most important to compare kernels to see if implemented improvements has been successful and behave as expected. A screenshot of a kernel investigated in NVIDIA Visual Profiler is given in Figure 3.4

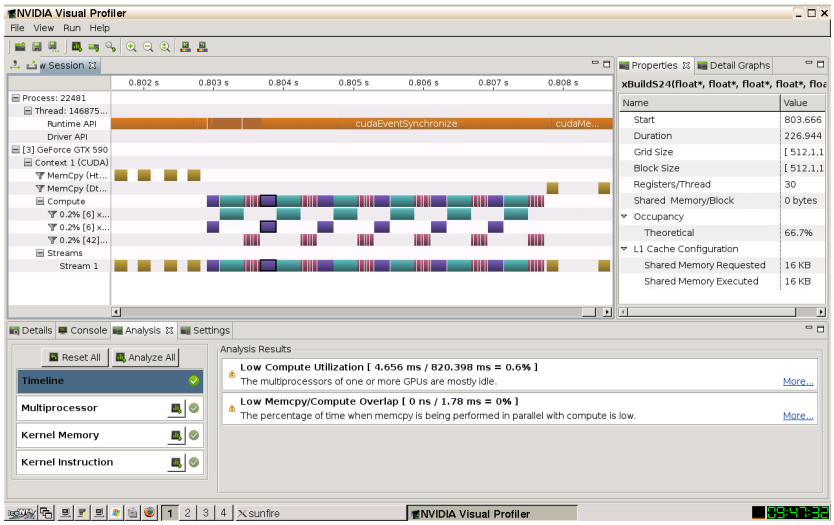


Figure 3.4: Screenshot of one of the implemented kernels run in NVIDIA Visual Profiler.

In Section 3.12 are shown, how useful performance metrics are calculated. The used counters can NVIDIA Visual Profiler provide.

3.12 Calculation of Performance Metrics

3.12.1 Identifying performance limiters: Memory or compute bound

It is beneficial to have some knowledge of what performance limiters the application has, simply to focus the optimization, but also to know how well one can expect the application to perform. Usually, there is differentiated between bandwidth or arithmetic limitations. A way to determine this one can determine the ratio comparisons instructions and memory bandwidth for the application and set it against the perfect balance for the given graphic card. The ratio instruction to bytes can be determined, cf. [48, p. 7], by

$$\frac{32 \cdot \text{instruction issued}}{128 \text{ bytes} \cdot \text{Global store transaction} + \text{L1 global load miss}} \tag{3.12.1}$$

Thus if we are higher than the perfect ratio (for the used graphic card) the application is likely arithmetic/compute bounded, lower it is memory bounded.

3.12.2 Theoretical bandwidth calculation

One can calculate the theoretical bandwidth for a given graphic card by using its hardware specifications, [32, sec. 5.2.1]

$$\text{Theoretical [GB/s]} = \frac{\text{Clock [MHz]} \cdot 10^6 \cdot (\text{Interface [Bits]}/8) \cdot \text{Rate}}{10^9} \tag{3.12.2}$$

3.12.3 Effective bandwidth

The effective bandwidth is determined by the number of bytes the application efficient reads and writes over time, hence cf. [32, sec. 5.2.2]

$$\text{Effective bandwidth} = \frac{\left(\text{Reads}_{\text{bytes}} + \text{Writes}_{\text{bytes}}\right)/1024^3}{\text{Time}_{\text{s}}} \tag{3.12.3}$$

The effective bandwidth can also be obtained by the NVIDIA Visual Profiler metrics: The Requested Global Load Throughput and The Requested Global Store Throughput. This metric is a very useful, since it can be used to investigate how a kernel perform and to see how well it utilize the hardware. Additional, it can be used to compare with the actual bandwidth to estimate how much bandwidth is wasted, see Section 3.5.1.

3.12.4 Divergent branches

The ratio of divergent branches is simply determined by the ratio of divergent branches and total branches. Since divergence within a warp causes serialization in execution should it obviously be avoided.

$$\text{Ratio of divergent branches} = \frac{\text{Number of divergent branches}}{\text{Number of total branches}} \cdot 100\% \tag{3.12.4}$$

3.12.5 Control flow divergence

The control flow divergences measure the percentage of thread instruction, which was not executed by all thread within the warp

$$\text{Control flow div.} = \frac{32 \cdot \text{inst. executed} - \text{threads inst. executed}}{32 \cdot \text{inst. executed}} \cdot 100\% \tag{3.12.5}$$

3.12.6 Replayed instructions

Replayed instruction measure the number of instruction that are issued by the hardware to the number of instructions that are to be executed by the kernel (percentage)

$$\text{Replayed Inst.} = \frac{\text{instructions issued} - \text{instruction executed}}{\text{instruction issued}} \cdot 100\% \tag{3.12.6}$$

3.13 Performance Optimization Strategies

In practice will some optimization steps have far greater impact than others depending on the type of problem. The key aspects that should always be taken into consideration is

- Maximizing parallel execution.
- Optimizing memory usage to achieve maximum memory bandwidth.
- Optimizing instruction usage to achieve maximum instruction throughput.

Which as a rule of thumb can be obtained by following optimization steps.

- Implement kernels with as much parallelism as possible.
- Make use global memory transactions are coalesced when possible.
- Minimize use of global memory, use faster on chip memory instead.
- Avoid warp divergence.
- Avoid bank conflicts.
- Achieve high occupancy.
- Make block sizes a multiple of warp size.

If these optimization step is performed can it be assumed that most crucial performance issues has been solved and the result should be a decent optimized kernel. It should be mentioned that it is important to compare performance to what is theoretically possible and use tools as the CUDA Occupancy Calculator and NVIDIA Visual Profiler.

Chapter 4

Numerical Formulation

This chapter contain a description of the shallow water equations, which are the equation used by MIKE 21 Flow Model to simulate flow and water level variations, see Section 4.1. These equation will be discretized and the derivation of the coefficients used to set up the system of equations will be described in detailed in Section 4.2.

4.1 Shallow Water Equations

The hydrodynamic model in the MIKE 21 Flow Model (MIKE 21 HD) is a numerical modelling system for simulation of water movements in lakes, estuaries, bays, coastal areas and seas. It simulates unsteady two-dimensional flows in one layer vertically homogeneous fluids.

The shallow water equations are a set of hyperbolic partial differential equations which model the propagation in incompressible fluids, under the condition that the vertical length scale is small compared to the horizontal length scale. I.e., the depth of the fluid are much smaller than the wave length, as we, e.g., see in lakes, bays and coastal areas, but also in the oceans when modelling the catastrophic tsunamis.

The equations are constructed from the theory of conservation of mass and momentum, and the partial differential equations that describe the flow and water level variations are as follows, (see Appendix A Table A.1 on page 130 for symbol description and unit specification.)

$$\frac{\partial \zeta}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} = \frac{\partial d}{\partial t} \tag{4.1.1}$$

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pp}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) + gh\frac{\partial \zeta}{\partial x} + \frac{gp\sqrt{p^2+q^2}}{C^2h^2} - \frac{1}{\rho_w}\left(\frac{\partial}{\partial x}(h\tau_{xx}) + \frac{\partial}{\partial y}(h\tau_{xy})\right) - \Omega_q - fVV_x + \frac{h}{\rho_w}\frac{\partial}{\partial x}(p_a) = 0 \tag{4.1.2}$$

$$\frac{\partial q}{\partial t} + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + gh\frac{\partial \zeta}{\partial y} + \frac{gq\sqrt{p^2+q^2}}{C^2h^2} - \frac{1}{\rho_w}\left(\frac{\partial}{\partial y}(h\tau_{yy}) + \frac{\partial}{\partial x}(h\tau_{xy})\right) - \Omega_p - fVV_y + \frac{h}{\rho_w}\frac{\partial}{\partial y}(p_a) = 0 \tag{4.1.3}$$

The time, t, in seconds and the two space coordinates, x and y, in meters are independent variables. The dependent variables are the surface elevation, ζ, in meters and the two-dimensional flux densities, p and q in $m^3/s/m$.

Figure 4.1 illustrate the interaction between the bathymetry (ground elevation), water surface elevation and water depth.

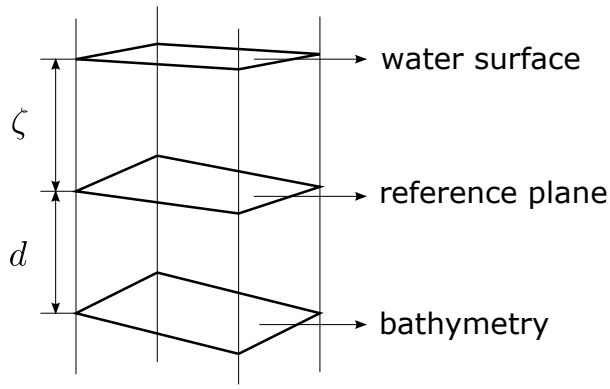


Figure 4.1: Overview of the bathymetry, water surface elevation and depth.

4.2 Introduction to Derivation of Discretization

In the following will we derive the numerical discretization of the inner points in the grid (points away from the coast) for the two-dimensional shallow water equations (4.1.1)-(4.1.3) described in Section 4.1.

The discretization is derived closely to the scientific documentation by DHI [21, chap. 3, 4 and 6], such that we obtain the same equations as used in MIKE 21 HD. However, the discretization will be explained in far greater details here. DHI has stated that their standard hydrodynamic simulations shall be implemented in the scheme. Therefore, some high order terms to obtain higher accuracy for short wave applications of the scheme have been neglected in this project. The truncation errors embedded in the finite difference approximation can be determined by the use of Taylor series expansions. Thus, we expect the error to be of 2nd order in Δx, i.e., $O(\Delta x^2)$. To clarify how the order of accuracy is obtained will a small example be given, which illustrates the approach, when analyzing the error in finite different approximation [16, chap. 1].

Assume we will approximate the first derivative of $u(x)$ using forward difference approximation. Applying Taylor series at point $\bar{x}$ we get

$$u(\bar{x}+h) = u(\bar{x}) + hu'(\bar{x}) + \frac{1}{2}h^2u''(\bar{x}) + \frac{1}{6}h^3u'''(\bar{x}) + O(h^4) \tag{4.2.1}$$

Thus applying (4.2.1) in forward difference approximation

$$D_+u(\bar{x}) = \frac{u(\bar{x}+h)-u(\bar{x})}{h} = u'(\bar{x}) + \frac{1}{2}hu''(\bar{x}) + \frac{1}{6}h^2u'''(\bar{x}) + O(h^3) \tag{4.2.2}$$

$u''(\bar{x})$ and $u'''(\bar{x})$ are fixed constant, since $\bar{x}$ is a fixed point. We see for sufficiently small h, that the error contribution will be dominated by the term $\frac{1}{2}hu''(\bar{x})$. Hence, the truncation error is expected to be of first order in h, $O(h)$.

We will derive the discretization of the mass and momentum equation in the x-direction. However, because the y-mass equation has influence on the centering of the x-mass equation will this derivation be considered too. Regarding the momentum equation will only x-momentum (4.1.2) be shown, since discretization of the y-momentum equations is performed in a similar manner.

4.2.1 Methods for discretization and solution scheme

MIKE 21 HD solves the shallow water equation on a staggered grid (see Figure 4.2) using several finite difference approximations schemes. Finite difference methods is used to obtain the numerical solution of the differential equations, which theory are assumes to be known. The used solution scheme is the Alternating Direction Implicit (ADI) method to solve the equations for mass and momentum conservation in the time domain for one row or column at a time,

in an alternating order. The resulting system of equations for each direction, respectively, a row or a column in the grid, result in a tri-diagonal matrix (see Section 4.5).

The ADI method is a finite difference method used for time discretization. The ADI method splits the finite difference equations into two stages, so-called sweeps (x- and y-sweep), per time step, where only one operator is taken implicitly at each stage. The equations are solved in both of the stages, i.e. first sweep the x-derivative is taken implicitly and y-derivative is taking explicitly and vice versa in the next sweep. In each sweep the equations are solved to update the flow and water level.

An advantage of using the ADI method is that the resulting system of equations, that needs to be solved for each row and column in the grid, are symmetric with nonzero elements only on the sub-, main- and super-diagonal, thus it is a sparse matrix. This special structure can be solved efficiently with tri-diagonal matrix algorithms like the Thomas algorithm. A further description of tri-diagonal solution algorithms can be found in Chapter 5.

4.2.2 Staggered grid in (x, y)-space

In Figure 4.2 is the staggered grid in (x,y)-space shown.

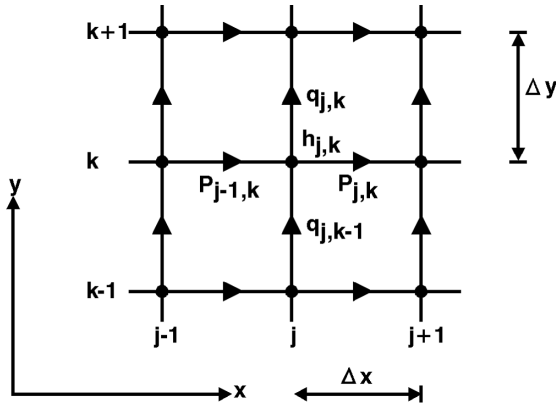


Figure 4.2: Staggered grid in (x,y)-space, [21, fig. 3.1].

The water depth, h, and water surface elevation, ζ, are located in-between the flux densities p and q.

This grid is used to derive the expression for the difference terms.

4.2.3 Time centering

The equations (4.1.1)-(4.1.3) are solved in one-dimensional sweeps in x- and y-direction. An x-sweep solves x-mass and x-momentum equations and consequently ζ is taken from n to $n+1/2$, p from n to $n+1$ and for terms involving q are values at $n-1/2$ and $n+1/2$ used. An y-sweep solves y-mass and y-momentum equations and consequently ζ is taking from $n+1/2$ to $n+1$, q from $n+1/2$ to $n+3/2$ and for terms involving p are the values at n and $n+1$ used.

Thereby is one time step achieved after one x- and one y-sweep, which solve the three equations (4.1.1)-(4.1.3). By adding the two sweeps we achieve a time centering of the various terms in the equations. The computational cycle is illustrated in Figure 4.3

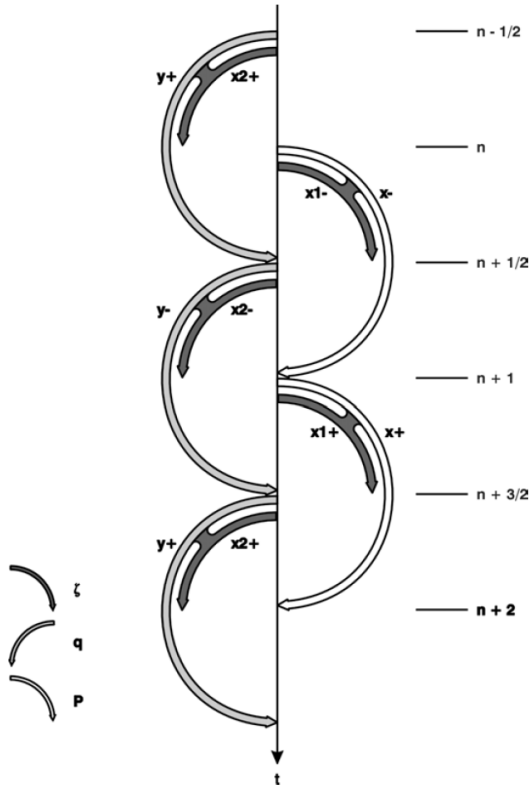


Figure 4.3: Time centering overview of computation cycle, [21, fig. 6.1].

We see that time centering at $n+1/2$ cannot be obtained by an x-sweep only. The y-mass has to be introduced before the centering is at $n+1/2$. In this way we obtain a computational cycle, where the sweeps together maintain the time centering.

4.3 Discretization of Mass Equations

We will derive the discretization of the mass equation from (4.1.1). Due to the time centering, described in Section 4.2.3, will both x- and y-mass equations be discretized, hence an x-sweep together with a y-mass equation will time centre the terms at $n + 1/2$.

As shown in Table A.1 on page 130 is d constant over time in this project. Therefore is $\partial d/\partial t = 0$, since the derivative of a constant is zero. Thus following mass equation of the 2D shallow water equations needs to be solved

$$\frac{\partial \zeta}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} = 0 \tag{4.3.1}$$

With the ADI method, backward finite difference approximation and the desired time centering at $n + 1/2$ in mind, we have the discretization of (4.3.1) in the x-direction, for which the grid notation is illustrated in Figure 4.2.

$$\begin{aligned} &2\frac{\zeta_{j,k}^{n+1/2} - \zeta_{j,k}^{n}}{\Delta t} + \frac{1}{2}\left(\frac{p_{j,k}^{n+1} - p_{j-1,k}^{n+1}}{\Delta x} + \frac{p_{j,k}^{n} - p_{j-1,k}^{n}}{\Delta x}\right) \\ &+\frac{1}{2}\left(\frac{q_{j,k}^{n+1/2} - q_{j,k-1}^{n+1/2}}{\Delta y} + \frac{q_{j,k}^{n-1/2} - p_{j,k-1}^{n-1/2}}{\Delta y}\right) = 0 \end{aligned} \tag{4.3.2}$$

where $\zeta_{j,k}^{n+1/2}$, $p_{j,k}^{n+1}$ and $p_{j-1,k}^{n+1}$ are the unknowns and the rest are known values from previews time steps.

In the same manner we have the discretization of the mass equation in the y-direction as

$$\begin{aligned} &2\frac{\zeta_{j,k}^{n+1} - \zeta_{j,k}^{n+1/2}}{\Delta t} + \frac{1}{2}\left(\frac{p_{j,k}^{n+1} - p_{j-1,k}^{n+1}}{\Delta x} + \frac{p_{j,k}^{n} - p_{j-1,k}^{n}}{\Delta x}\right) \\ &+\frac{1}{2}\left(\frac{q_{j,k}^{n+3/2} - q_{j,k-1}^{n+3/2}}{\Delta y} + \frac{q_{j,k}^{n+1/2} - p_{j,k-1}^{n+1/2}}{\Delta y}\right) = 0 \end{aligned} \tag{4.3.3}$$

where $\zeta_{j,k}^{n+1}$, $q_{j,k}^{n+3/2}$ and $q_{j,k-1}^{n+3/2}$ are the unknowns and the rest are known values from previews time steps.

The water depth is updated after each sweep, based on the water surface elevation and bathymetry, thus after an x-sweep: $h^{n+1/2} = \zeta^{n+1/2} - d$ and after an y-sweep: $h^{n+1} = \zeta^{n+1} - d$.

4.4 Discretization of Momentum Equations

In this section we will derive the discretization of the x-momentum equation from (4.1.2). The discretization of the y-momentum equations is performed in a similar manner and is therefore not shown.

As shown in Table A.1 on page 130 are some of the terms in (4.1.2) neglect in this project, thus following x-momentum equation is solved

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pp}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) + gh\frac{\partial \zeta}{\partial x} + \frac{gp\sqrt{p^2+q^2}}{C^2h^2} = 0 \tag{4.4.1}$$

We will deduce the discretization by approximating the derivatives using finite difference approximations by considering each term one at a time.

Note that all terms in (4.4.1) are time centered at $n+1/2$ and space centered at $p_{j,k}$ in the staggered grid.

4.4.1 Discretization of the time derivation term

Applying forward finite difference approximation to the time derivative term, we have

$$\left.\frac{\partial p}{\partial t}\right|_{j,k} \approx \frac{p_{j,k}^{n+1} - p_{j,k}^{n}}{\Delta t} \tag{4.4.2}$$

4.4.2 Discretization of the convective momentum term

The difference form for the convective momentum term is based on taking one of the p at time n and the other one at $n+1$. Performing central difference approximation using step size $\Delta x/2$, we define

$$\hat{D}_0 u(x) = \frac{1}{\Delta x}\left[u\left(x + \frac{\Delta x}{2}\right) - u\left(x - \frac{\Delta x}{2}\right)\right]$$

Thus we obtain the discretization in the x-direction, with reference to the grid notation illustrated in Figure 4.4, as

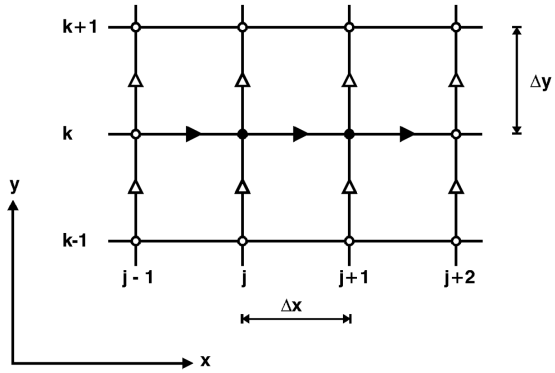


Figure 4.4: Grid notation for the convective term in the x-momentum equation, [21, fig. 4.5].

$$\frac{\partial}{\partial x}\left(\frac{pp}{h}\right)\bigg|_{j,k} \approx \frac{1}{\Delta x}\left[\frac{p^{n+1}_{j+1/2,k}\cdot p^{n}_{j+1/2,k}}{h^{n}_{j+1,k}} - \frac{p^{n+1}_{j-1/2,k}\cdot p^{n}_{j-1/2,k}}{h^{n}_{j,k}}\right] \tag{4.4.3}$$

In order to obtain an approximation to, e.g., $p_{j+1/2}$ are following expression used

$$p_{j+1/2} \triangleq \frac{1}{2}\left(p_{j+1} + p_j\right)$$

Hence we can rewrite (4.4.3) to

$$\begin{aligned}\frac{\partial}{\partial x}\left(\frac{pp}{h}\right)\bigg|_{j,k} \approx \frac{1}{\Delta x}\Bigg(&\frac{p^{n+1}_{j+1,k}+p^{n+1}_{j,k}}{2}\cdot\frac{p^{n}_{j+1,k}+p^{n}_{j,k}}{2}\cdot\frac{1}{h^{n}_{j+1,k}}\\ &-\frac{p^{n+1}_{j,k}+p^{n+1}_{j-1,k}}{2}\cdot\frac{p^{n}_{j,k}+p^{n}_{j-1,k}}{2}\cdot\frac{1}{h^{n}_{j,k}}\Bigg)\end{aligned} \tag{4.4.4}$$

where $p^{n+1}_{j-1,k}$, $p^{n+1}_{j,k}$ and $p^{n+1}_{j+1,k}$ are the unknowns and the rest is known. Note that (4.4.4) results in a penta-diagonal matrix system of difference equations, since we now have three unknown flux densities and later will get two unknown water surface elevations. Therefore, we reduce the system to a tri-diagonal form by local substitution before applying a tri-diagonal matrix solver. A description on how the local substitution is performed can be found in Section 6.3.1.

Furthermore are the difference form in (4.4.4) used for flow at low Froude numbers[1]. The MIKE 21 HD solution procedure to maintain robustness of

[1] The Froude numbers is defined as the water velocity divided by the water wave propagation velocity.

the numerical solution for flow at high Froude numbers, is simply to introduce a weighting scalar in (4.4.4) which is dependent of the Froude number. The weighting scalar, which is determined for each time step for each grid point, is applied to the convective momentum terms. Thus the numerical dissipation is only used at grid points where flows at high Froude numbers are present. However, this will not be implemented in our project.

4.4.3 Discretization of the cross momentum term

The cross momentum term is approximated by the same approach as the convective momentum term in 4.4.2. However, there will be some variation in the indexing, since the grid points for the flux densities p and q do not overlap.

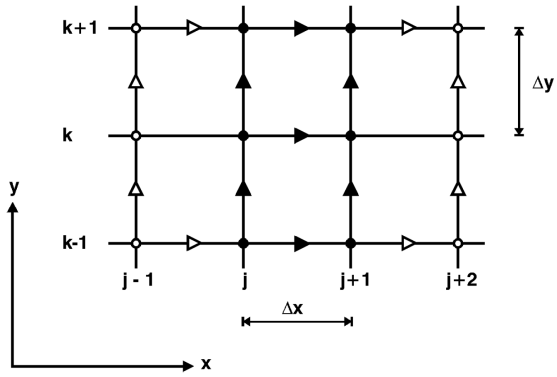


Figure 4.5: Grid notation for the cross term in the x-momentum equation, [21, fig. 4.6].

Thus we have the discretization in the x-direction, with referring to the grid notation illustrated in Figure 4.5, as

$$\frac{\partial}{\partial y}\left(\frac{pq}{h}\right)\bigg|_{j,k} \approx \frac{1}{\Delta y}\left(\frac{p^n_{j,k+1}+p^n_{j,k}}{2}v^{n+1/2}_{j+1/2,k} - \frac{p^n_{j,k}+p^n_{j,k-1}}{2}v^{n+1/2}_{j+1/2,k-1}\right) \tag{4.4.5}$$

where

$$v^{n+1/2}_{j+1/2,k} = \frac{q^{n+1/2}_{j,k}+q^{n+1/2}_{j+1,k}}{2}\cdot\frac{1}{\bar{h}_1} \tag{4.4.6}$$

$$v^{n+1/2}_{j+1/2,k-1} = \frac{q^{n+1/2}_{j,k-1}+q^{n+1/2}_{j+1,k-1}}{2}\cdot\frac{1}{\bar{h}_2} \tag{4.4.7}$$

$\hat{h_1}$ and $\hat{h_2}$ are the interior points of the water depth approximated by the arith-

metic average of the four influence surrounding points, thus

$$\bar{h_1} = \frac{h^n_{j,k} + h^n_{j,k+1} + h^n_{j+1,k} + h^n_{j+1,k+1}}{4} \tag{4.4.8}$$

$$\bar{h_2} = \frac{h^n_{j,k-1} + h^n_{j,k} + h^n_{j+1,k-1} + h^n_{j+1,k}}{4} \tag{4.4.9}$$

4.4.4 Discretization of the gravity term

By applying forward finite difference approximation to the gravity term and approximating h as the arithmetic average, we have

$$gh\frac{\partial \zeta}{\partial x}\bigg|_{j,k} \approx g\frac{h^n_{j,k} + h^n_{j+1,k}}{2}\frac{\zeta^{n+1/2}_{j+1,k} - \zeta^{n+1/2}_{j,k}}{\Delta x} \tag{4.4.10}$$

Recalling that $h^n_{j,k} = \zeta^n_{j,k} - d_{j,k}$. Thus the terms have been linearized and discretized. In this project are $g = 9.81$ m/s^2 used.

4.4.5 Discretization of the resistance term

The bed shear stress is approximated using the Chézy formula. The Chézy coefficient is in MIKE 21 HD determined by using the Manning coefficient. Using this coefficient, the stress in x-direction is expressed as

$$\frac{gp\sqrt{p^2+q^2}}{C^2h^2} \tag{4.4.11}$$

Hence when linearizing the resistance term the approximation becomes

$$\frac{gp\sqrt{p^2+q^2}}{C^2h^2} \approx \frac{gp^{n+1}_{j,k}\sqrt{\hat{p}^2+\hat{q}^2}}{C^2\hat{h}^2} \tag{4.4.12}$$

where

$$\hat{p} = p^n_{j,k}$$

$$\hat{q} = \frac{1}{8}\left(q^{n-1/2}_{j,k} + q^{n-1/2}_{j+1,k} + q^{n-1/2}_{j,k-1} + q^{n-1/2}_{j+1,k-1} + q^{n+1/2}_{j,k} + q^{n+1/2}_{j+1,k} + q^{n+1/2}_{j,k-1} + q^{n+1/2}_{j+1,k-1}\right)$$

$$\hat{h} = \sqrt{\frac{2}{\left(\frac{1}{h^n_{j,k}}\right)^2 + \left(\frac{1}{h^n_{j+1,k}}\right)^2}}$$

The Chézy coefficient, C, is determined as mentioned above from the Manning coefficient, M, as

$$C = M \cdot \hat{h}^{1/6}$$

4.5 Setting Coefficients for an x-sweep

In this section will the coefficients for an x-sweep be determined and the matrix system, which is determined and solved in each row in the grid, will be constructed, by using the performed discretization of the mass and momentum equation in x-direction. Notice that the construction of the system of equations for the y-direction is performed in a similar manner.

Performing an x-sweep results in a penta-diagonal matrix

$$a p_{j-1,k}^{n+1} + b\zeta_{j,k}^{n+1/2} + c p_{j,k}^{n+1} = d \tag{4.5.1}$$

$$l^* p_{j-1,k}^{n+1} + a^* \zeta_{j,k}^{n+1/2} + b^* p_{j,k}^{n+1} + c^* \zeta_{j+1,k}^{n+1/2} + r^* p_{j+1,k}^{n+1} = d^* \tag{4.5.2}$$

where the coefficients a,b,c,d and l^*, a^*,b^*,c^*,r^*,d^* are known expressions from x-mass and x-momentum respectively.

Notice that only every other element in the penta-diagonals have values and therefore the matrix can be efficiently reduced to a tri-diagonal matrix.

4.5.1 Coefficient for mass equation

The expressions for a,b,c and d will be derived in the following by isolating the unknown $p_{j-1,k}^{n+1}, \zeta_{j,k}^{n+1/2}$ and $p_{j,k}^{n+1}$ in the x-mass equation (4.3.2), thus

$$\begin{aligned} 2\frac{\zeta_{j,k}^{n+1/2} - \zeta_{j,k}^{n}}{\Delta t} + \frac{1}{2}\left(\frac{p_{j,k}^{n+1} - p_{j-1,k}^{n+1}}{\Delta x} + \frac{p_{j,k}^{n} - p_{j-1,k}^{n}}{\Delta x}\right) \\ + \frac{1}{2}\left(\frac{q_{j,k}^{n+1/2} - q_{j,k-1}^{n+1/2}}{\Delta y} + \frac{q_{j,k}^{n-1/2} - p_{j,k-1}^{n-1/2}}{\Delta y}\right) = 0 \end{aligned}$$

By extending with $2\Delta x$, we obtain

$$\begin{aligned} 4\frac{\Delta x}{\Delta t}\left(\zeta_{j,k}^{n+1/2} - \zeta_{j,k}^{n}\right) + p_{j,k}^{n+1} - p_{j-1,k}^{n+1} + p_{j,k}^{n} - p_{j-1,k}^{n} \\ + \frac{\Delta x}{\Delta y}\left(q_{j,k}^{n+1/2} - q_{j,k-1}^{n+1/2} + q_{j,k}^{n-1/2} - q_{j,k-1}^{n-1/2}\right) = 0 \end{aligned}$$

Rearranging, we obtain

$$\begin{aligned} -p_{j-1,k}^{n+1} + 4\frac{\Delta x}{\Delta t}\zeta_{j,k}^{n+1/2} + p_{j,k}^{n+1} \\ = p_{j-1,k}^{n} - p_{j,k}^{n} - \frac{\Delta x}{\Delta y}\left(q_{j,k}^{n+1/2} - q_{j,k-1}^{n+1/2} + q_{j,k}^{n-1/2} - q_{j,k-1}^{n-1/2}\right) + 4\frac{\Delta x}{\Delta t}\zeta_{j,k}^{n} \end{aligned}$$

Thus we have the expressions for a,b,c and d as

$$\begin{aligned} a &= -1 \\ b &= 4\frac{\Delta x}{\Delta t} \\ c &= 1 \\ d &= p^n_{j-1,k} - p^n_{j,k} - \frac{\Delta x}{\Delta y}\left(q^{n+1/2}_{j,k} - q^{n+1/2}_{j,k-1} + q^{n-1/2}_{j,k} - q^{n-1/2}_{j,k-1}\right) + 4\frac{\Delta x}{\Delta t}\zeta^n_{j,k} \end{aligned} \tag{4.5.3}$$

4.5.2 Coefficient for momentum equation

The expressions for l^*, a^*,b^*,c^*, r^* and d^* will be derived in the following by inserting the discretized terms and isolating the unknown $p^{n+1}_{j-1,k}$, $\zeta^{n+1/2}_{j,k}$, $p^{n+1}_{j,k}$, $\zeta^{n+1/2}_{j+1,k}$ and $p^{n+1}_{j+1,k}$ in the x-momentum equation (4.1.2), thus will be derived to

$$\begin{aligned} &\frac{p^{n+1}_{j,k} - p^n_{j,k}}{\Delta t} + \frac{1}{\Delta x}\left(\frac{p^{n+1}_{j+1,k} + p^{n+1}_{j,k}}{2} \cdot \frac{p^n_{j+1,k} + p^n_{j,k}}{2} \cdot \frac{1}{h^n_{j+1,k}}\right. \\ &\qquad \left. - \frac{p^{n+1}_{j,k} + p^{n+1}_{j-1,k}}{2} \cdot \frac{p^n_{j,k} + p^n_{j-1,k}}{2} \cdot \frac{1}{h^n_{j,k}}\right) \\ &+ \frac{1}{\Delta y}\left(\frac{p^n_{j,k+1} + p^n_{j,k}}{2} v^{n+1/2}_{j+1/2,k} - \frac{p^n_{j,k} + p^n_{j,k-1}}{2} v^{n+1/2}_{j+1/2,k-1}\right) \\ &+ g\frac{h^n_{j,k} + h^n_{j+1,k}}{2}\frac{\zeta^{n+1/2}_{j+1,k} - \zeta^{n+1/2}_{j,k}}{\Delta x} + \frac{g p^{n+1}_{j,k}\sqrt{\hat{p}^2 + \hat{q}^2}}{C^2\hat{h}^2} = 0 \end{aligned}$$

The 1st and 4th term (time derivation term and the gravity term) are split, while the 2nd term (convective momentum term) is factored into groups for, respectively, $p^{n+1}_{j-1,k}$, $p^{n+1}_{j,k}$ and $p^{n+1}_{j+1,k}$, thus

$$\begin{aligned} &\frac{1}{\Delta t}p^{n+1}_{j,k} - \frac{1}{\Delta t}p^n_{j,k} + \frac{p^n_{j+1,k}\, p^n_{j,k}}{4\Delta x h^n_{j+1,k}} p^{n+1}_{j+1,k} \\ &+ \left(\frac{p^n_{j+1,k}\, p^n_{j,k}}{4\Delta x h^n_{j+1,k}} - \frac{p^n_{j,k}\, p^n_{j-1,k}}{4\Delta x h^n_{j,k}}\right) p^{n+1}_{j,k} - \frac{p^n_{j,k}\, p^n_{j-1,k}}{4\Delta x h^n_{j,k}} p^{n+1}_{j-1,k} \\ &+ \frac{1}{\Delta y}\left(\frac{p^n_{j,k+1} + p^n_{j,k}}{2} v^{n+1/2}_{j+1/2,k} - \frac{p^n_{j,k} + p^n_{j,k-1}}{2} v^{n+1/2}_{j+1/2,k-1}\right) \\ &+ \frac{g}{2\Delta x}\left(h^n_{j,k} + h^n_{j+1,k}\right)\left(\zeta^{n+1/2}_{j+1,k} - \zeta^{n+1/2}_{j,k}\right) + \frac{g\sqrt{\hat{p}^2 + \hat{q}^2}}{C^2\hat{h}^2} p^{n+1}_{j,k} = 0 \end{aligned}$$

Further factoring and rearranging the different terms, we obtain

$$\begin{aligned}
&-\frac{p^n_{j,k}\,p^n_{j-1,k}}{4\Delta x h^n_{j,k}}p^{n+1}_{j-1,k} - \frac{g}{2\Delta x}\left(h^n_{j,k}+h^n_{j+1,k}\right)\zeta^{n+1/2}_{j,k} \\
&+\left[\frac{1}{\Delta t}+\frac{g\sqrt{\hat{p}^2+\hat{q}^2}}{C^2\hat{h}^2}+\left(\frac{p^n_{j+1,k}\,p^n_{j,k}}{4\Delta x h^n_{j+1,k}}-\frac{p^n_{j,k}\,p^n_{j-1,k}}{4\Delta x h^n_{j,k}}\right)\right]p^{n+1}_{j,k} \\
&+\frac{g}{2\Delta x}\left(h^n_{j,k}+h^n_{j+1,k}\right)\zeta^{n+1/2}_{j+1,k}+\frac{p^n_{j+1,k}\,p^n_{j,k}}{4\Delta x h^n_{j+1,k}}p^{n+1}_{j+1,k} \\
=\frac{1}{\Delta t}p^n_{j,k}&-\frac{1}{\Delta y}\left(\frac{p^n_{j,k+1}+p^n_{j,k}}{2}v^{n+1/2}_{j+1/2,k}-\frac{p^n_{j,k}+p^n_{j,k-1}}{2}v^{n+1/2}_{j+1/2,k-1}\right)
\end{aligned}$$

Thus we have the expressions for l^*, a^*, b^*, c^*, r^* and d^* as

$$\begin{aligned}
l^* &= -\frac{p^n_{j,k}\,p^n_{j-1,k}}{4\Delta x h^n_{j,k}} \\
a^* &= -\frac{g}{2\Delta x}\left(h^n_{j,k}+h^n_{j+1,k}\right) \\
b^* &= \frac{1}{\Delta t}+\frac{g\sqrt{\hat{p}^2+\hat{q}^2}}{C^2\hat{h}^2}+\frac{p^n_{j+1,k}\,p^n_{j,k}}{4\Delta x h^n_{j+1,k}}-\frac{p^n_{j,k}\,p^n_{j-1,k}}{4\Delta x h^n_{j,k}} \\
c^* &= \frac{g}{2\Delta x}\left(h^n_{j,k}+h^n_{j+1,k}\right) \\
r^* &= \frac{p^n_{j+1,k}\,p^n_{j,k}}{4\Delta x h^n_{j+1,k}} \\
d^* &= \frac{p^n_{j,k}}{\Delta t}-\frac{p^n_{j,k+1}+p^n_{j,k}}{2\Delta y}v^{n+1/2}_{j+1/2,k}+\frac{p^n_{j,k}+p^n_{j,k-1}}{2\Delta y}v^{n+1/2}_{j+1/2,k-1}
\end{aligned} \tag{4.5.4}$$

4.5.3 Set up the penta-diagonal matrix system

The derived coefficients for mass and momentum in x-direction (see above Section 4.5.1 and 4.5.2) are set up in following manner

$$
\begin{bmatrix}
b_1 & c_1 & & & & & & & & \\
a_1^* & b_1^* & c_1^* & r_1^* & & & & & & \\
 & a_2 & b_2 & c_2 & & & & & & \\
 & l_2^* & a_2^* & b_2^* & c_2^* & r_2^* & & & & \\
 & & & a_3 & b_3 & c_3 & & & & \\
 & & & l_3^* & a_3^* & b_3^* & c_3^* & r_3^* & & \\
 & & & & \ddots & \ddots & \ddots & \ddots & \ddots & \\
 & & & & & & a_{n-1} & b_{n-1} & c_{n-1} & \\
 & & & & & & l_{n-1}^* & a_{n-1}^* & b_{n-1}^* & c_{n-1}^* \\
 & & & & & & & & a_n & b_n
\end{bmatrix}
\begin{bmatrix}
\zeta_1^{n+1/2} \\ p_1^{n+1} \\ \zeta_2^{n+1/2} \\ p_2^{n+1} \\ \zeta_3^{n+1/2} \\ p_3^{n+1} \\ \vdots \\ \zeta_{n-1}^{n+1/2} \\ p_{n-1}^{n+1} \\ \zeta_n^{n+1/2}
\end{bmatrix}
$$

$$
= \begin{bmatrix} d_1 & d_1^* & d_2 & d_2^* & d_3 & d_3^* & \dots & d_{n-1} & d_{n-1}^* & d_n \end{bmatrix}^T
$$

This system of equations is set up for each row in the grid of an x-sweep and each column in the grid of an y-sweep. A local elimination will be performed to reduce it from a penta- to a tri-diagonal matrix system. Hereafter, a tri-diagonal matrix algorithm will be applied to obtain the solution of the system. How this is done will be described in Chapter 5.

Chapter 5

Tri-diagonal Solver Algorithms

In this chapter will three different tri-diagonal matrix solver algorithms used in this project be described. The algorithms are the Thomas algorithm, Parallel Cyclic Reduction (PCR) and a hybrid version combining Cyclic Reduction and Parallel Cyclic Reduction (CR-PCR) as presented in [47]. All three algorithms will be examined from a general theoretical point of view and will later be modified and optimized to the specific problem presented by DHI. For simplicity, when illustrating matrices, will empty spaces represent values that are zero. Throughout the Chapter will modified equations and coefficients be labelled with $'$, hence e' represents a modified e.

5.1 Tri-diagonal matrix

A tri-diagonal matrix is a matrix which only has values in the sub-, main- and super-diagonal. The tri-diagonal matrix is very interesting for this project, since each row or column in the grid will return a system that has to be solved of the form

$$Ax = d \tag{5.1.1}$$

where A is a tri-diagonal matrix. Hence the system will look like

$$\begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & \ddots & & \\ & & \ddots & \ddots & c_{n-2} & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{5.1.2}$$

The tri-diagonal matrix contains very few non-zero elements, thus it is a sparse matrix. Therefore, one should take advantage of this special structure. For instance, is it a good idea only to store a sparse representation of the matrix. Normally this would involve four vectors; one for each of the diagonals and one for the right hand side of the system. However, in this project will the vectors be split up even more. Every other element will be stored in separate vectors, hence we have 8 vectors of half the system size. The reason for this will be described further in Section 6.2.

MIKE 21 HD solves a lot of tri-diagonal systems. Consequently, it is important to investigate different solver algorithms to have effective applications. The different algorithms used in this project will be described in the following.

5.2 The Thomas Algorithm

The solution algorithm used by DHI in MIKE 21 HD is the Thomas algorithm. It is a very efficient serial algorithm that requires $2n$ steps to solve a $n \times n$ tri-diagonal system. The algorithm is serial in that sense that all computations depends on the previous calculations. This implies that in the standard version as implemented by DHI no computations can be performed in parallel.

The algorithm is divided into two phases; a forward elimination phase, where all the values in the sub-diagonal are eliminated and a backwards substitution phase, where the solution is obtained.

5.2.1 Forward elimination

In the forward elimination phase the system is updated as follows

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & \text{if } i = 1, \\ \dfrac{c_i}{b_i - c'_{i-1}a_i} & \text{if } i = 2,3,...,n-1 \end{cases} \tag{5.2.1}$$

and

$$d'_i = \begin{cases} \dfrac{d_i}{b_i} & \text{if } i = 1, \\ \dfrac{d_i - d'_{i-1}a_i}{b_i - c'_{i-1}a_i} & \text{if } i = 2,3,...,n \end{cases} \tag{5.2.2}$$

It is clear that all updates (except the initial step) are directly dependent on the preceding calculation. Thus this part of the algorithm is serial and must take n steps, where only a single c and d are updated in each step. In each update the element in the sub-diagonal are eliminated and the row is divided by b_i to obtain one in the main-diagonal. This means that the updating is basically sparse row

operations which makes the updating a lot more efficient than normal Gaussian elimination. A half way reduced system is illustrated in (5.2.3).

$$\begin{bmatrix} 1 & c'_1 & & & & & \\ & 1 & c'_2 & & & & \\ & & \ddots & \ddots & & & \\ & & & 1 & c'_{n-2} & & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} & \\ & & & & a_n & b_n & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d'_1 \\ d'_2 \\ \vdots \\ d'_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix} \tag{5.2.3}$$

It should be noticed that the denominator in (5.2.1) and (5.2.2) when updating c_i and d_i are identical, thus one of the divisions can be avoided. Pseudo code for the forward elimination is given as We see that only one division is needed

Algorithm 1 : Forward elimination

1: $tmp = \dfrac{1}{b_1}$
2: $c'_1 = c_1 \cdot tmp$
3: $d'_1 = d_1 \cdot tmp$
4: **for** $i = 2$ to n **do**
5: $\quad tmp = \dfrac{1}{b_i - c'_{i-1} \cdot a_i}$
6: $\quad c'_i = c_i \cdot tmp$
7: $\quad d'_i = \left(d_i - d'_{i-1} \cdot a_i\right) \cdot tmp$
8: **end for**

for each step in the loop. Clearly, very few operations have to be done in each step, which shows how simple, yet effective, the algorithm is.

5.2.2 Backwards substitution

After the forward elimination the resulting system is a matrix with only a main-diagonal consist of ones and an super-diagonal as shown in (5.2.4).

$$\begin{bmatrix} 1 & c'_1 & & & & & \\ & 1 & c'_2 & & & & \\ & & \ddots & \ddots & & & \\ & & & 1 & c'_{n-2} & & \\ & & & & 1 & c'_{n-1} & \\ & & & & & 1 & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d'_1 \\ d'_2 \\ \vdots \\ d'_{n-2} \\ d'_{n-1} \\ d'_n \end{bmatrix} \tag{5.2.4}$$

It is clear that x_n directly can be read, since it is equal to d_n. Hereafter, it is trivial to deduced the renaming unknowns. Formally the resulting solution is

obtained by

$$x_i = \begin{cases} d'_n & \text{if } i = n, \\ d'_i - c'_i \cdot x_{i+1} & \text{if } i = n-1, n-1 \ldots 1 \end{cases} \tag{5.2.5}$$

Which easily can be calculated by following pseudo code. Again it is clear that

Algorithm 2 : Forward elimination

1: $x_n = d'_n$
2: **for** $i = n - 1$ to 1 **do**
3: $\quad x_i = d'_i - c'_i \cdot x'_{i+1}$
4: **end for**

all updates (except the initial step) are directly dependent on the preceding calculation. Therefore, the algorithm is serial and will take n steps to determine all the unknowns. Further we see that the algorithm is very simple and very few instructions have to be performed in each step. Hence the Thomas algorithm is a very efficient serial algorithm performing only $2n$ steps and $O(n)$ operations. The only drawback is that it is only possible to use a single processor to solve the system, when running the algorithm as described. This indicate that if only a single system has to be solved this might be the best solution scheme to run on a CPU, since only few processors are available. Unlike on a GPU, this method is not the fastest, since there would be a lot of processors idle. Nevertheless, a scenario there the algorithm may be preferred on a GPU would be if many systems had to be solved simultaneously, which is the case in the project. Thus it will be investigated further.

5.3 Parallel Cyclic Reduction

Parallel Cyclic Reduction (PCR) is another algorithm that solves tri-diagonal systems. It does much more work than the Thomas algorithm, but the computations can be calculated in parallel. Thereby, it has fewer algorithmic steps if enough processors are available. The algorithm is chosen, because it is found to be the fastest of the standard algorithms presented in [47].

In each step of the algorithm all elements in both the sub- and super-diagonal are eliminated simultaneously. However, due to the structure of the matrix this results in a new sub- and super-diagonal, just moved further away from the main-diagonal as illustrated in (5.3.1).

$$
\begin{bmatrix}
b_1 & c_1 & & & & & & \\
a_2 & b_2 & c_2 & & & & & \\
 & a_3 & b_3 & c_3 & & & & \\
 & & a_4 & b_4 & c_4 & & & \\
 & & & a_5 & b_5 & c_5 & & \\
 & & & & a_6 & b_6 & c_6 & \\
 & & & & & a_7 & b_7
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
b'_1 & & c'_1 & & & & \\
 & b'_2 & & c'_2 & & & \\
a'_3 & & b'_3 & & c'_3 & & \\
 & a'_4 & & b'_4 & & c'_4 & \\
 & & a'_5 & & b'_5 & & c'_5 \\
 & & & a'_6 & & b'_6 & \\
 & & & & a'_7 & & b'_7
\end{bmatrix}
\Rightarrow
$$

$$
\begin{bmatrix}
b''_1 & & & c''_1 & & & \\
 & b''_2 & & & c''_2 & & \\
 & & b''_3 & & & c''_3 & \\
 & & & b''_4 & & & \\
a''_5 & & & & b''_5 & & \\
 & a''_6 & & & & b''_6 & \\
 & & a''_7 & & & & b''_7
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
b'''_1 & & & & & & \\
 & b'''_2 & & & & & \\
 & & b'''_3 & & & & \\
 & & & b'''_4 & & & \\
 & & & & b'''_5 & & \\
 & & & & & b'''_6 & \\
 & & & & & & b'''_7
\end{bmatrix}
\tag{5.3.1}
$$

In the first step the diagonals are moved one element away from the main-diagonal. In the second step they are moved two elements further away. In the third step four elements further and so on. For each step, the distance the diagonals are moved, is doubled. Until they are completely removed from the system. This results in $\log_2(n)$ steps with n operations in each step, which can be performed in parallel. So even though the algorithm performs much more work than the Thomas algorithm, it can run faster given that there are enough parallel processing power available.

The elements are updated in the following manner

$$a'_i = -a_{i-s} \cdot k_1 \tag{5.3.2}$$

$$c'_i = -c_{i+s} \cdot k_2 \tag{5.3.3}$$

$$b'_i = b_i - c_{i-s} \cdot k_1 - a_{i+1} \cdot k_2 \tag{5.3.4}$$

$$d'_i = d_i - d_{i-s} \cdot k_1 - d_{i+1} \cdot k_2 \tag{5.3.5}$$

$$k_1 = \begin{cases} 0 & \text{if } i \text{ is the first equation of a system,} \\ \dfrac{a_i}{b_{i-s}} & \text{otherwise} \end{cases} \tag{5.3.6}$$

$$k_2 = \begin{cases} 0 & \text{if } i \text{ is the last equation of a system,} \\ \dfrac{c_i}{b_{i+s}} & \text{otherwise} \end{cases} \tag{5.3.7}$$

where s is the stride, which is equal to one in the first step, is doubled in each step. As it was the case for the Thomas algorithm the updating in PCR is just

sparse row operations, where the row above is used to eliminate a and the row below is used to eliminate c. This means that for each row to be updated values from two other rows are used (except from the first and last row, where only one other row are used). The workflow of the algorithm is illustrated in Figure 5.1.

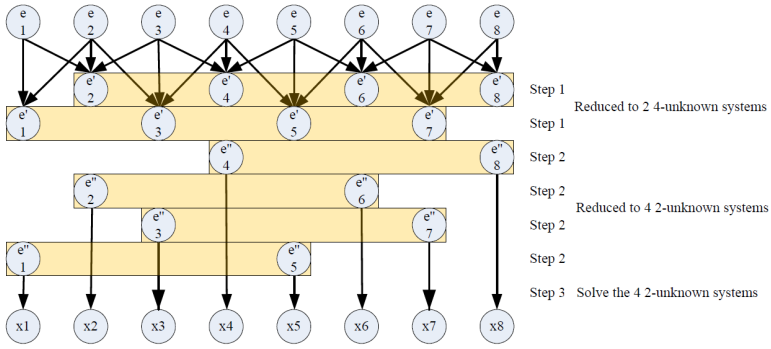


Figure 5.1: Workflow of PCR algorithm in the 8-unknown case, [47, fig. 2]. Equations are labelled $e1$-$e8$. Equations in a yellow rectangle form a independent system. Arrows are omitted in step 2 for clarity.

It can be realized that updating in the described way, actually reduces the tri-diagonal system into two new tri-diagonal systems of half the size in each step. After the first iteration, every other element is an independent system. After the second iteration every fourth element is an independent systems and so on, as illustrated in Figure 5.1. This can also be illustrated by rearranging the rows and columns of the matrix as shown in (5.3.8).

$$\begin{bmatrix} b'_1 & c'_1 & & & & & \\ & b'_2 & & c'_2 & & 0 & \\ a'_3 & & b'_3 & & c'_3 & & \\ & a'_4 & & b'_4 & & c'_4 & \\ & & a'_5 & & b'_5 & & c'_5 \\ & 0 & & a'_6 & & b'_6 & \\ & & & & a'_7 & & b'_7 \end{bmatrix} \Rightarrow \begin{bmatrix} b'_1 & c'_1 & & & & & \\ a'_3 & b'_3 & c'_3 & & & 0 & \\ & a'_5 & b'_5 & c'_5 & & & \\ & & a'_7 & b'_7 & & & \\ & & & & b'_2 & c'_2 & \\ & 0 & & & a'_4 & b'_4 & c'_4 \\ & & & & & a'_6 & b'_6 \end{bmatrix}$$

$$\begin{bmatrix} b''_1 & & & c''_1 & & & \\ & b''_2 & & & c''_2 & & \\ & & b''_3 & 0 & & c''_3 & \\ & & & b''_4 & & & \\ a''_5 & & 0 & & b''_5 & & \\ & a''_6 & & & & b''_6 & \\ & & a''_7 & & & & b''_7 \end{bmatrix} \Rightarrow \begin{bmatrix} b''_1 & c''_1 & & & & & \\ a''_5 & b''_5 & & & & 0 & \\ & & b''_2 & c''_2 & & & \\ & & a''_6 & b''_6 & & & \\ & & & & b''_3 & c''_3 & \\ & 0 & & & a''_7 & b''_7 & \\ & & & & & & b''_4 \end{bmatrix} \tag{5.3.8}$$

This algorithm is a lot more complicated than the Thomas algorithm and as mentioned it performs a lot more work. However, it could be more beneficial to run on a GPU, since it allow more threads to work simultaneously.

5.4 Cyclic Reduction + Parallel Cyclic Reduction Hybrid

The last tri-diagonal matrix solver algorithm, that will be used, is a hybrid version of Cyclic Reduction (CR) and PCR. The algorithm is chosen partly because it is found to be the fastest algorithm presented in [47]. Furthermore it is expected to be especially useful for this assignment, which will be discussed further in Section 8.4.2.

CR is very similar to PCR. The only difference is that PCR produces more smaller tri-diagonal systems and solves these simultaneously, while CR focuses on solving only one of the resulting systems from each step. Hence the updating is exactly the same as in PCR, but CR halves the work that has to be computed in each step, thus the resulting system is of half the system size. This implies that the algorithm needs a backwards substitution phase, where the solution to the remaining systems are obtained. The updating in the backwards substitution phase is given as

$$x_i = \frac{d'_i - a'_i \cdot x_{i-1} - c'_i \cdot x_{i+1}}{b_i} \tag{5.4.1}$$

It takes CR $2\log_2(n)$ algorithmic steps to finish, because of the backwards substitution phase. This is twice as many as PCR, but each step requires a lot less work; the first step takes only $n/2$ operations, the second step take $\frac{n}{4}$ operations and so on, while PCR makes n operations for all the steps. The workflow of CR is shown in Figure 5.2.

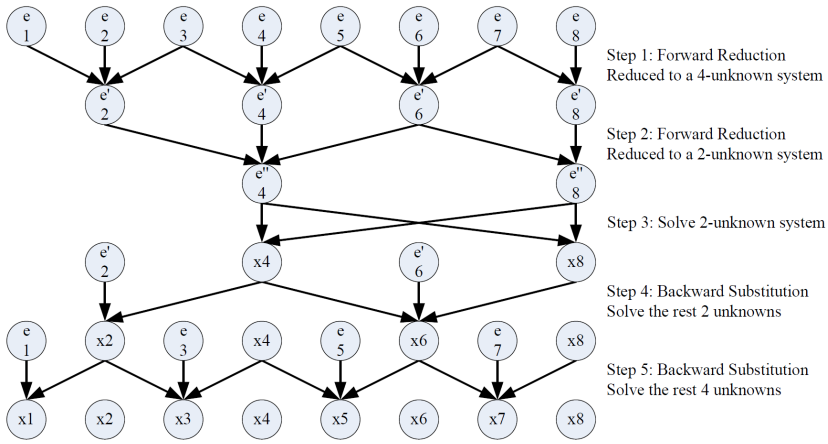


Figure 5.2: Workflow of CR algorithm in the 8-unknown case, [47, fig. 1]. Equations are labelled $e1$-$e8$.

In Table 5.1 are the algorithmic operations and algorithmic steps for all the algorithms shown.

Algorithm	Total arithmetic operations	Algorithmic steps	Parallelism
TA	$2n$	$2n$	1
PCR	$12n\log_2(n)$	$\log_2(n)$	n
CR	$17n$	$2\log_2(n)-1$	$n/2^k$
CR-PCR	$17(n-m)+12m\log_2(m)$	$2\log_2(n)-\log_(m)-1$	$n/2^k$ or m

Table 5.1: Algorithmic operations and algorithmic steps for the different algorithms where n is the system size, m is the intermidiate system size and k represænts the step. Both n and m is assumed to be a power of 2.

It is clear that the CR performs much less work than PCR but that PCR takes fewer steps which is why it is reasonable to think that the hybrid version is a good compromise.

In a perfect setting, where unlimited parallel processing power is available, it is oblivious that PCR will outperform CR by a factor of 2. However, in almost all cases, there will be some restrictions. Therefore, it is likely that it could be beneficial to run a few CR steps to reduce the size of the system, before switching to PCR to solve the smaller system. This approach is the so called CR-PCR hybrid. The workflow of the hybrid algorithm can be seen in Figure 5.3.

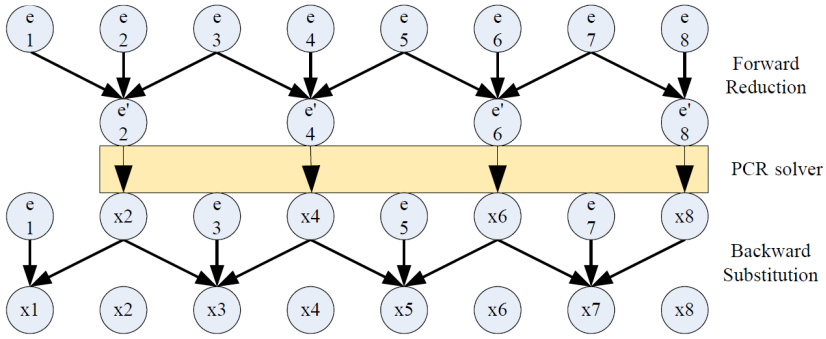


Figure 5.3: Workflow of CR-PCR hybrid algorithm in the 8-unknown case, [47, fig. 4]. Equations are labelled $e1$-$e8$. Details about PCR are omitted see Figure 5.1.

When using the hybrid approach it should be clear that every other element in the original system only will be used for the first and the last step of the backwards substitution in CR. Thus these elements are not modified throughout the algorithm. This is a very attractive quality, when considering the system that needs to be solved in this project, which will be shown later in Section 8.4.2.

Chapter 6

Sequential C Implementation

This chapter contains the outline of MIKE 21 HD and a description of our sequential C implementation, developed to maintain same structure as MIKE 21 HD. The C implementation will be used as a comparison for the parallel CUDA C implementations in terms of correctness and performance. Therefore, in Section 6.5 will we investigate whether the C application is representative for MIKE 21 HD programmed in the programming language FORTRAN. Additional will the C application be verified and validated.

The entire C application with and without inflow can be found in Part II in the Source Code Booklet.

6.1 Comprehension into MIKE 21 HD

MIKE 21 HD was first developed in the 1980s and has since then gone though a lot of improvements. It is implemented in FORTRAN, which neither of us has any experience with. The application is very complex, with more over than 8,000 lines of code. Furthermore, is a lot of the performed approximations/discretization of the mathematical expressions in the code (especially on the boundaries) based on experience rather than exact mathematical approximations. This means it was difficult to derive the discretization of the shallow water equations, since our C implementation shall obtain an identical solution given the same input as MIKE 21 HD. An internal scientific documentation for MIKE 21 HD, provided by DHI, was incomprehensible for extraneous and unfortunately contained mistakes and errors. For these reasons, just understanding the existing program for not to mention implementing an identical version, was a very extensive task. Hence a lot of the time in the project was going with that.

6.1.1 Outlining the MIKE 21 HD flow operate

The main tasks in MIKE 21 HD is to

- Build the tri-diagonal matrix for a given system (a row or column) in the grid.
- Solve the system and return the solution.

When referring to *a system* is it either one row or column in the grid as illustrated in Figure 6.1.

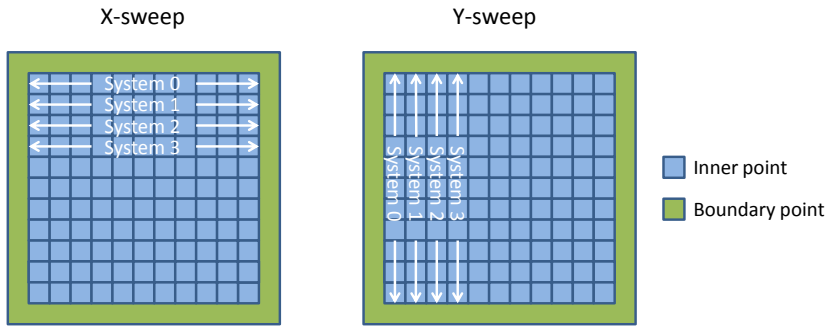


Figure 6.1: Explansion of *a system* in the grid.

Hence a lot of tri-diagonal systems need to be build and solved for each time step. Notice that for a sweep, are each systems independent of each other. Further are the calculations of the coefficients independent of each other when building the tri-diagonal matrix. However, solving each element in the system depend on the solution of the other elements.

As illustrated in Figure 6.1 are all points/cells in the grid not equal. Thus inner points refer to water points and outer-points/boundary point refer to points in the boundary which can either be a land point of contain inflow. The handling of these special cells is described in Section 6.4.

6.2 Data Structures

In order to run the application is it obviously necessary to have access to the ζ, p, q and bathymetry for all grid points to different time steps. In MIKE 21 HD are these values stored in 2-dimensional arrays, but in our C application will we store them in a linear memory structure. Since we still want to access the grid based on coordinates are a simple 2D $\leftrightarrow$ 1D mapping used as shown in (6.2.1).

$$(x,y) \rightarrow idx : idx = x + y \cdot n \tag{6.2.1}$$

where n is the dimension of the grid. Notice that the mapping is chosen based on memory access in C/C++ style (row-wise).

MIKE 21 HD uses a unique way of storing the tri-diagonal system. It is common that a tri-diagonal system is represented as only four vectors; one to store each of the diagonals and one for the right hand side. However, DHI have chosen a different approach, where the mass and momentum equations are stored in separated vectors. This means they store the system in 8 vectors each of half the system size. The reason for this lies in the building of the system, since mass and momentum equations is set up in very different ways. Thus we have the vectors: `ama`, `amo`, `bma`, `bmo`, `cma`, `cmo`, `dma` and `dmo` which stores the tri-diagonal system as shown in (6.2.2).

$$\begin{bmatrix} \mathtt{bma}_1 & \mathtt{cma}_1 & & & & & & & \\ \mathtt{amo}_1 & \mathtt{bmo}_1 & \mathtt{cmo}_1 & & & & & & \\ & \mathtt{ama}_2 & \mathtt{bma}_2 & \mathtt{cma}_2 & & & & & \\ & & \mathtt{amo}_2 & \mathtt{bmo}_2 & \mathtt{cmo}_2 & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & \mathtt{ama}_{n-1} & \mathtt{bma}_{n-1} & \mathtt{cma}_{n-1} & & \\ & & & & & \mathtt{amo}_{n-1} & \mathtt{bmo}_{n-1} & \mathtt{cmo}_{n-1} & \\ & & & & & & \mathtt{ama}_n & \mathtt{bma}_n & \end{bmatrix} x$$
$$= \begin{bmatrix} \mathtt{dma}_1, \mathtt{dmo}_1, \mathtt{dma}_2, \mathtt{dmo}_2, \cdots, \mathtt{dma}_{n-1}, \mathtt{dmo}_{n-1}, \mathtt{dma}_n \end{bmatrix}^T \tag{6.2.2}$$

Thereby are every other element in the diagonals stored in the vectors labelled `ma` (mass equations) and the other elements are stored in the vectors labelled `mo` (momentum equations). From the derivation of the terms, we have that `ama`, `bma` and `cma` are all defined as constant independent of the sweep.

Notice that for each row or column the resulting tri-diagonal system are build and solved before the next system is build. This means that it is only necessary to allocate memory for a single tri-diagonal system and reuse those vectors throughout the application. Because we want our implementation to relate to MIKE 21 HD and since this seem like a fairly efficient data structure will it be implemented in exactly the same manner in the C implementation.

6.3 Development of the Sequential C Implementation

Implementing the entire MIKE 21 HD would be an impossible task given the time frame of this project. Thus the programs developed in this project only contain the core functionalities of MIKE 21 HD. In the best possible manner we have maintained the same structure, call tree and chosen data structures to insure the most identical code compared to MIKE 21 HD. The reason it is interesting to implement a sequential version of MIKE 21 HD in C, is to gain a fair comparison and to verify the correctness of the parallel CUDA C implementations. This means that it will be used to verify the correctness of the CUDA implementation and to indicate how much time the CUDA implementation takes compared to a serial version.

6.3.1 Implementation details

The implemented C application is divided into subroutines to maintain the same workflow as MIKE 21 HD, but also to obtain a robust and good code. The workflow of the program is illustrated in Figure 6.2. The y-sweep is performed in exactly same manner as the x-sweep (only transposed), thus it will not be described further.

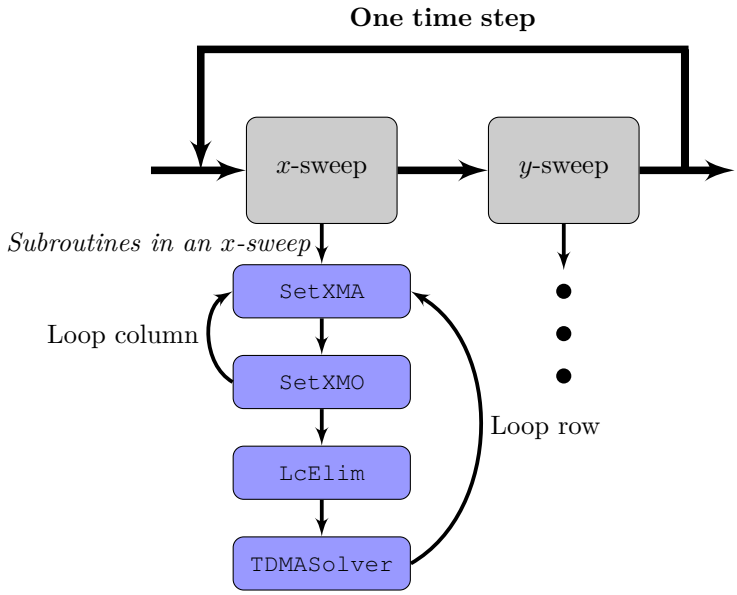


Figure 6.2: Workflow of the C application.

The application is called by the routine `ShallowWater_gold.c`. Here are first the vectors `ama`, `amo`, `bma`, `bmo`, `cma`, `cmo`, `dma`, `dmo`, `rmo`, `lmo` and `x` allocated to contain the mass and momentum equations for the penta-diagonal matrix and to store the determined solution. The x- and y-sweeps are called through the subroutines `xsweepGold` and `ysweepGold` inside a `for`-loop, which run over the specified number of time steps.

The x-sweep contain a nested `for`-loop; the outer loop run through all rows and the inner loop run through all columns in the system. In the inner loop are the subroutines `SetXMAGold` and `SetXMOGold` called, which sets up the mass and momentum equations, respectively. The coefficients for these equations can be seen in Section 4.5 in (4.5.3) and (4.5.4).

When all mass and momentum equations for one row is set up, i.e., when the inner loop terminates, the subroutine `LcElimGold` is executed. `LcElimGold` reduces the penta-diagonal system into a tri-diagonal system. This is done by running a `for`-loop eliminating all coefficient in the sub-diagonal and then another loop eliminating the super-diagonal. The eliminations are performed by sparse row operations. After this are the `TDMASolverGold` subroutine called. This routine uses the Thomas algorithm to solve the tri-diagonal system, as described in Section 5.2. This is done by first running a `for`-loop, which eliminates the sub-diagonal and then a `for`-loop that performs a back-substitution and stores the solution in the vector `x`. At last the solution is written from x to ζ and p in a `for`-loop. When all rows in the system have been handled this way is the outer loop terminated and an x-sweep is performed on the system.

An pseudo code is provided to give further overview of the program.

Algorithm 3 : Fluid Simulation

Input bathymetry and initial water surface elevation and flux in x- and y-direction.

1: **for** $i = 1$ to number of time steps **do**
2: **for** $k = 1$ to number of rows **do**
3: **for** $j = 1$ to number of columns **do**
4: Build the penta-diagonal system by set up the mass and momentum equations for the k^{th} row.
5: **end for**
6: Reduce the penta-diagonal matrix into a tri-diagnoal matrix by local elimination.
7: Solve the tri-diagonal matrix system.
8: Return the calculated solution to the water surface elevation, $\zeta^{n+1/2}$, and flux densities in x-direction, p^{n+1}.
9: **end for**
10: ...
11: Run an y-sweep, similar to the above x-sweep.
12: **end for**

6.3.2 Complexity of C application

As described, a sweep contain an outer loop with 6 loops inside, which runs over all rows/columns. The two sweeps are called for each time step, thus we end up with a complexity (disregarding any of the actual work done in the loops) of

$$\begin{aligned} T(n,t) &= t \cdot 2 \cdot n \cdot 6 \cdot n = 12 \cdot t \cdot n^2 \\ &= O\left(t \cdot n^2\right) \end{aligned}$$

Hence the complexity is quadratic in terms of the system size n and linear in the number of time steps t. Therefore, solving a large system over many time steps can quickly become a very extensive task.

6.4 Handling of Boundary Conditions

We have first developed a simple version of the program. It is simple in the sense that it can only calculate grids with coast around the boundary and where all inner grid points are water, like a swimming pool. Obviously, the grid points near the boundary has to be handled differently than grid points surrounded entirely by water. This implies the need of boundary conditions.

MIKE 21 HD handles boundary conditions by having several 2-dimensional arrays, where the element value represent what the surrounding grid points are. Thereby they can get the boundary value and though some `if`-statements change the parameter and expression as desired. A simple example is given in Listing 6.1, where the fraction of code handles west end of chain, specially if it is near a land point.

Listing 6.1: Example of how MIKE 21 HD handle boundary conditions with land.

```
1  ...
2
3  C          ------------------------------------------------------
4  C              -- One-point wide chain
5  C          ------------------------------------------------------
6  C            treated above (east end of chain !!)
7
8             if (side.ne.1) call ehndle(-18,jfst,k,area)
9
10 C          ------------------------------------------------------
11 C          -- Get hand on the boundary value
12 C          ------------------------------------------------------
13            bndv = bndval(no,k,2)
14
15 . . .
16
17 C             ---------------------------------------
18 C             -- Update boundary at full time step
19 C             -- for yfab
```

```
20 c              ------------------------------------------
21                  bndval(no,k,1) = bndval(no,k,1) - xbnd
22
23               ama(jfst) = 0.0
24               bma(jfst) = 1.0
25               cma(jfst) = 0.0
26               dma(jfst) = bndv
27               bmo(jfst) = bmo(jfst) + lmo(jfst)
28               lmo(jfst) = 0.
29 c             west end finished
30               return
31
32             elseif (irr .eq. 7) then
33 c         ------------------------------------------------
34 c         -- Open flux boundary
35 c         ------------------------------------------------
36               if (no .eq. 0) call ehndle(-18,jfst,k,area)
37                   amo(jfst) = 0.0
38                   bmo(jfst) = 1.0
39                 cmo(jfst) = 0.0
40                 dmo(jfst) = bndval(no,k,1)
41                 lmo(jfst) = 0.0
42                 rmo(jfst) = 0.0
43 c               dummy mass.
44                   ama(jfst) = 0.0
45                   bma(jfst) = 1.0
46                 cma(jfst) = 0.0
47                 dma(jfst) = 0.0
48             else
49               if (no .ne. 0) call ehndle(-13,jfst,k,area)
50 c             ------------------------------------------------
51 c             -- If we arrive here we have an west boundary
52 c             -- problem.
53 c             ------------------------------------------------
54             endif
55 c        else
56 c             open boundary in subarea:
57 c         call ehndle(-18,jfst,k,area)
58          endif
59
60 . . .
```

The way that we handle boundary conditions is also by applying `if`-statements that makes sure that flux and water levels is never read from the boundary. So when calculating the point (j,k) and $\zeta_{j,k+1}$ is in the boundary then the value should not be used. DHI have instead arbitrarily chosen to use the value in $\zeta_{j,k}$. This is illustrated in Listing 6.2, where `bL` is the bathymetry which is equal to `lh` if the element is a land point.

Listing 6.2: Example of how the C application handles boundary condition with land.

```
f1 = (qNp05[j+km1] + qNp05[jp1+km1]);
if (bL[j+km1]==lh)
{
    f1 /= (zetaN[jk] + zetaN[jp1+nY] + zetaN[jk] + zetaN[jp1+km1]
           - bL[jk] - bL[jp1+nY] - bL[jk] - bL[jp1+km1]);
}
else if (bL[jp1+km1]==lh)
{
    f1 /= (zetaN[jk] + zetaN[jp1+nY] + zetaN[j+km1] + zetaN[jp1+nY]
           - bL[jk] - bL[jp1+nY] - bL[j+km1] - bL[jp1+nY]);
}
else
{
    f1 /= (zetaN[jk] + zetaN[jp1+nY] + zetaN[j+km1] + zetaN[jp1+km1]
           - bL[jk] - bL[jp1+nY] - bL[j+km1] - bL[jp1+km1]);
}
```

The implementation, where the fraction of code is from, can be seen in Section 5.4 in the Source Code Booklet.

The complexity of the program has been extended further to allow inflow from the boundary. This add more conditions into the program, since a boundary with inflow again should be handled specially. This is because the boundary is not considered to change in the same way as the inner points in the grid.

A simple example of how MIKE 21 HD handles this, is given in Listing 6.3, which are continuation code from Listing 6.1. We see, as mention, how they take care of the boundary value by using the 2-dimensional array, thus the integer value represent both what kind of element it is and what the surrounding elements are.

Listing 6.3: Example of how MIKE 21 HD handles boundary condition with flux.

```
...

c          ------------------------------------------------------
c              -- One-point wide chain
c          ------------------------------------------------------
c            treated above (east end of chain !!)

           if (side.ne.1) call ehndle(-18,jfst,k,area)

c          ------------------------------------------------------
c          -- Get hand on the boundary value
c          ------------------------------------------------------
           bndv = bndval(no,k,2)

. . .

c             ---------------------------------------
```

```
18 c              -- Update boundary at full time step
19 c              -- for yfab
20 c              ------------------------------------------
21                  bndval(no,k,1) = bndval(no,k,1) - xbnd
22
23               ama(jfst) = 0.0
24               bma(jfst) = 1.0
25               cma(jfst) = 0.0
26               dma(jfst) = bndv
27               bmo(jfst) = bmo(jfst) + lmo(jfst)
28               lmo(jfst) = 0.
29 c             west end finished
30               return
31
32            elseif (irr .eq. 7) then
33 c         ------------------------------------------------------
34 c         -- Open flux boundary
35 c         ------------------------------------------------------
36               if (no .eq. 0) call ehndle(-18,jfst,k,area)
37                   amo(jfst) = 0.0
38                   bmo(jfst) = 1.0
39                 cmo(jfst) = 0.0
40                 dmo(jfst) = bndval(no,k,1)
41                 lmo(jfst) = 0.0
42                 rmo(jfst) = 0.0
43 c               dummy mass.
44                   ama(jfst) = 0.0
45                   bma(jfst) = 1.0
46                 cma(jfst) = 0.0
47                 dma(jfst) = 0.0
48              else
49               if (no .ne. 0) call ehndle(-13,jfst,k,area)
50 c             ------------------------------------------------------
51 c             -- If we arrive here we have an west boundary
52 c             -- problem.
53 c             ------------------------------------------------------
54              endif
55 c        else
56 c             open boundary in subarea:
57 c         call ehndle(-18,jfst,k,area)
58          endif
59
60 . . .
```

We handle the inflow in the same manner as before, thus we extended the program with `if`-statements, which handle the special conditions. An example is given in Listing 6.4, which illustrate the way we handle inflow from west.

Listing 6.4: Example of how C application handels boundary condition with flux.

```
. . .

if (k == 0 || k == sizeY+1)   // calculate on bnd, in case of water
    level inflow
{
    for(j=1; j<sizeX; j++)
    {
        ama[j] = 0.0;
        bma[j] = 1.0;
        cma[j] = 0.0;

        if (bL[j+nY] < lh)
        {
            dma[j] = wlbnd;
        } else {
            dma[j] = zetaN[j+nY];
        }
        amo[j] = 0.0;
        bmo[j] = 1.0;
        cmo[j] = 0.0;
        dmo[j] = 0.0;
    }
    ama[j] = 0.0;
    bma[j] = 1.0;
    cma[j] = 0.0;

    if (bL[j+nY] < lh)
    {
        dma[j] = wlbnd;
    } else {
        dma[j] = zetaN[j+nY];
    }
}
else                          // calculate on "water grid"
{
    if (bL[nY] < lh)   // if inflow from west
    {
        bcs = 1;
        ama[0] = 0.0;
        bma[0] = 1.0;
        cma[0] = 0.0;
        dma[0] = wlbnd;

        // Set x momentum equations.
    }

. . .
```

The implementation, where the fraction of code is from, can be seen in Section 6.2 in the Source Code Booklet.

6.5 Investigating Performance of MIKE 21 HD and C Application

In the following will MIKE 21 HD and the developed C application be profiled to investigate performance bottlenecks. This is done to check if the C application is representative for MIKE 21 HD. The profiling and performance comparison are performed on DHIs system[1].

6.5.1 Profiling MIKE 21 HD

It is interesting to investigate what the performance bottlenecks of MIKE 21 HD are to better understand the program and get an idea of what might become problem later on. For this reason has a profiling of MIKE 21 HD been performed by DHI for a 512×512 grid over 1000 time steps. It is done using the AQtime profiler and a screen shot from the result can be seen in Figure 6.3.

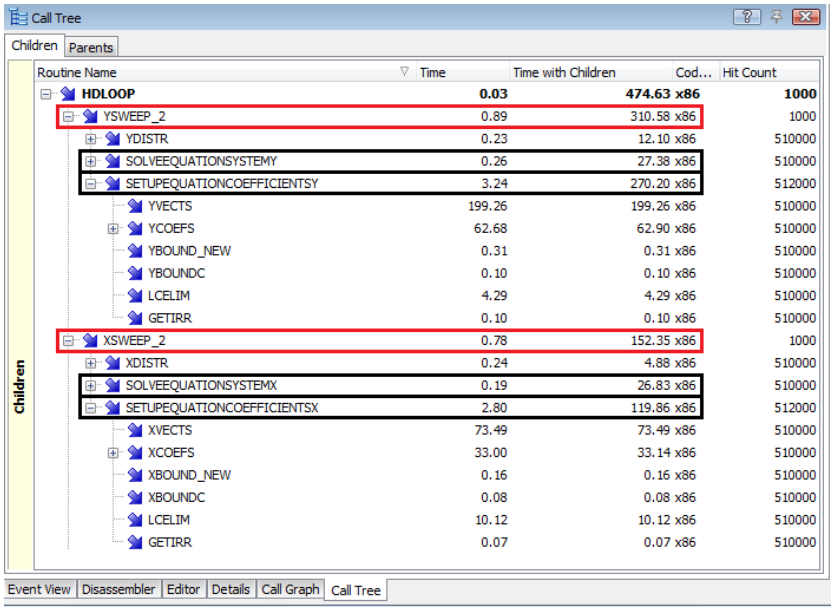


Figure 6.3: Profiling using AQtime of MIKE 21 HD on a 512×512 grid with 1000 time steps performed by DHI. Specification of the test system is given in Appendix B Table B.3 page 132.

From the screen shot is two things immediately clear; the y-sweep is twice as slow as the x-sweep and building the system takes up ∼84% of the total runtime. Solving the system takes approximately the same time for both sweeps, but

[1]System specification is available in Appendix B Table B.3 page 132

setting up the systems is a lot slower for the y-sweep. Since the x- and y-sweep must be assumed to perform roughly the same work is it likely that the y-sweep does not utilize the cache lines properly when accessing elements in the memory. It turns out that we will see a similar phenomenon in our parallel CUDA C implmentation, as described in section Section 8.2.2 and 8.3.1.

6.5.2 Profiling the C implementation

For a comparison to the MIKE 21 HD profiling have we also profiled our C implementation. A screen shot can be seen in Figure 6.4

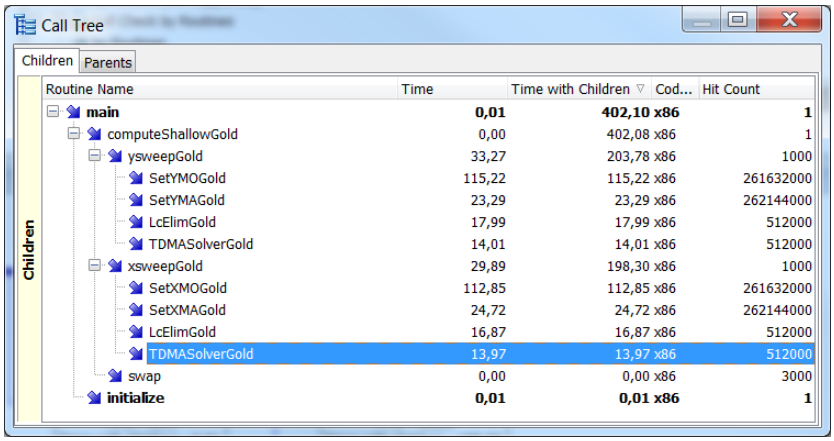


Figure 6.4: Profiling using AQtime Standard vers. 7 of the C implementation on a 512×512 grid with 1000 time steps. Specification of the test system is given in Appendix B Table B.3 page 132.

From our implementation, we see that the two sweeps take roughly the same amount of time which differs from MIKE 21 HD. However, building the system like MIKE 21 HD by far take the longest, with ∼70% of the compute time.

6.5.3 Performance comparison of MIKE 21 HD and C implementation

It is important that our C implementation is representative for MIKE 21 HD, since we later want to compare it to the time of our parallel CUDA C implementations. MIKE 21 HD is a lot more complex than our sequential version, so from that it would seems that our version would be faster. On the other hand MIKE 21 HD has been optimized a lot more. To make sure that our C implementation is representative for MIKE 21 HD, is a comparison between the runtime for different system sizes performed, see Table 6.1.

System size	128×128	256×256	512×512
MIKE 21 HD	1.030 s	4.290 s	17.784 s
C implementation	1.060 s	4.352 s	17.580 s

Table 6.1: Execution time of MIKE 21 HD and the C implementation for different system sizes over 100 time steps. Specification of the test system is given in Appendix B Table B.3 page 132.

From this it is clear that they perform very similar. In fact, they are almost identical. As expected, the time scales whit $O(n^2)$ of system size.

Thus our implemented sequential C implementation is representative for MIKE 21 HD and the obtained performance results will be a realistic measure for what DHI can expect to achieve.

6.6 Verification

It is expected that the solution obtained by the numerical approximation will converge to the analytical solution when decreasing the step size, Δx. Thus the accuracy will increase for lowering the step size. In fact, from the derivations described in Section 4.2, it is expected that the truncation errors embedded in the finite difference approximation should be described by

$$\|\epsilon\|_\infty \leq O\left(\Delta x^2\right) \tag{6.6.1}$$

where ϵ is the error of the approximation. Hence the discretization approximation has 2nd order of accuracy. In order to verify whether it holds for the application, can we compare the obtained results with an analytical solution to the equations

$$\frac{\partial \zeta}{\partial t} + h \cdot \frac{\partial p}{\partial x} = 0 \tag{6.6.2}$$

$$\frac{\partial p}{\partial t} + g \cdot \frac{\partial \zeta}{\partial x} = 0 \tag{6.6.3}$$

given as

$$\zeta(x,t) = \frac{H}{2} \cdot \cos(k \cdot x) \cdot \cos(\omega \cdot t) \tag{6.6.4}$$

$$p(x,t) = \frac{H}{2} \cdot \frac{\omega}{k \cdot h} \cdot \sin(k \cdot x) \cdot \sin(\omega \cdot t) \tag{6.6.5}$$

where the water depth is assumed constant to $h = 0.05$, the wave height and length is $H = 0.01$ and $L = 8$, respectively. Thus the wave amplitude is $H/2$ and the wave has $h/L = 0.00625$ and wave number $k = 2\pi/L$.

The verification cannot cover all the terms in the numerical approximation, since the solution is one-dimensional and does not cover flat bottom. For this

reason, when testing how the error converges, will a simpler and only one-dimensional numerical approximation be used. In Figure 6.5 is the truncations error as function of discretization parameter Δx for a small Δt shown

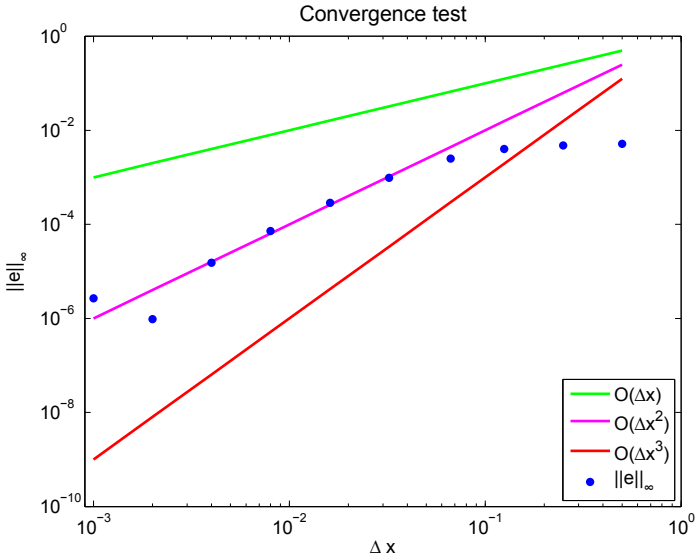


Figure 6.5: Convergence test. The truncation error follow 2nd order of convergent.

As expected, we see that the truncation error decreases as step size, Δx, decrease. The numerical model is convergent when $\Delta x \lessapprox 10^{-1.18}$ and hereafter follow the line for second order convergence perfectly. Thereby the truncation error behave as expected.

6.7 Validation

For validation, we make use of the results return by MIKE 21 HD, since DHI have required that we develop a parallel solution scheme which obtain the same result as MIKE 21 HD. However, as mention, MIKE 21 HD is an old and very worked through application which is why we can assume that it simulates the "real" world correctly.

The validation is applied throughout the development to insure that the sequential C implementation always calculate the same solution as MIKE 21 HD. The validation is performed by comparing all elements in both water surface elevation, ζ, and flux density in both direction, p and q by taking $\|e\|_\infty$, where e is the error vector for the difference between the solution obtained by MIKE 21 HDand the one from the C implementation [16, sec. A.3]. Thus the C implementation return the same results as MIKE 21 HD given the same input.

Figure 6.6-6.9 show a simulation of a wave started at the center of the grid. The simulation is run with $dx = 10$, $dy = 10$ and $dt = 2$.

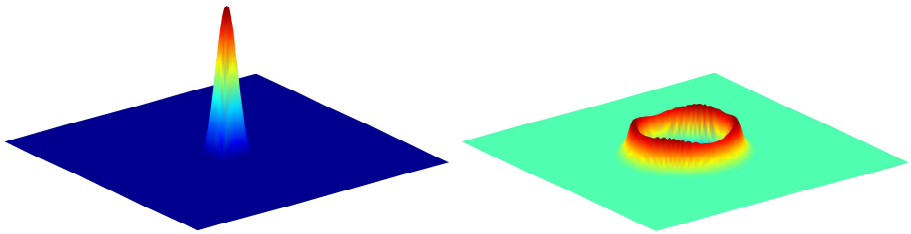


Figure 6.6: Illustration of the initial wave and how it has spread after 10 time steps.

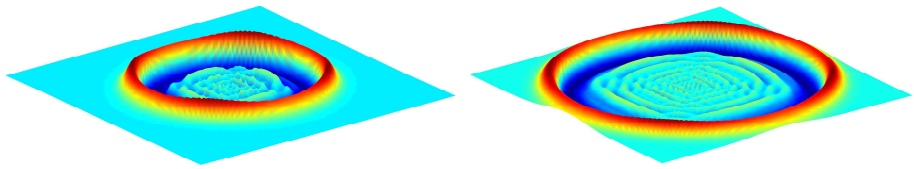


Figure 6.7: Illustration of the wave after 20 and 30 time steps respectively.

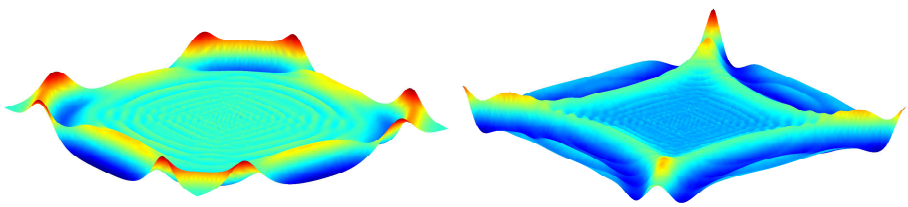


Figure 6.8: Illustration of the wave after 40 and 50 time steps respectively.

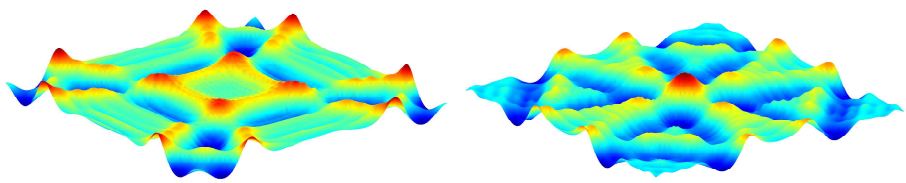


Figure 6.9: Illustration of the wave after 60 and 70 time steps respectively.

It is clear to see that the simulations behave as expected and since we are have landboundaries all around the grid are the waves rebound on the edge and return back towards the center.

Chapter 7

Parallel CUDA C Implementation

In this chapter there will be given an overview of the parallel CUDA C implementations. This involves an overview of the developed parallel approaches, the optimization strategy and an investigating of the limiting factors for the applications. Additional will the test environment be outlined and it will be explained how the massive parallel applications are validated.

The entire CUDA C application can be found in Part I in the Source Code Booklet.

It is important for DHI that the parallel CUDA C implementation bears some resemblance to the original C implementation and thereby also to MIKE 21 HD. This is because it makes it easier for DHI to incorporate the parallel CUDA C application into the existing MIKE 21 HD, cf. the original project description provided by DHI (see Appendix C page 133). For this reason the originally used workflow of the C application and the way of handling the data structures will be used in the CUDA implementation too. Thus the 10 vectors (`ama`, `amo`, `bma`, `bmo`, `cma`, `cmo`, `dma`, `dmo`, `lmo` and `rmo`) will be used to store the penta-diagonal matrix system. See Figure 6.2 for illustration of workflow at which some `for`-loops will be removed in the parallel CUDA C applications.

The application, like MIKE 21 HD, will be divided into the subroutines (device functions) `SetMA`, `SetMO`, `LcElim` and `TDMASolver`, which sets up mass- and momentum equations, reduce the penta-diagonal to a tri-diagonal matrix system and in the end solve the system and store the obtained results. However, since the purpose of this project is to investigate performance bottlenecks and optimize these to obtain a faster parallel solution scheme, are the application divided into 4 kernel calls per simulation time step. This is done to make it

easier to determine where the bottlenecks are located. The 4 kernels are called `xBuild`, `xSolve`, `yBuild` and `ySolve`, where `x` and `y` determines whether it is an x- or an y-sweep. The `Build` kernels contain the device functions `SetMA` and `SetMO`, which sets up the mass and momentum equations. The `Solve` kernels contain the device functions `LcElim` and `TDMASolver`, which reduce the penta-diagonal matrix system into a tri-diagonal matrix system in preparation for subsequently solving the system of equations.

7.1 Parallel Approaches

In Section 6.1, we have that each tri-diagonal system can be build and solved independently. Furthermore, can all coefficients for a given system can be calculated independently. This gives two different levels of parallelism to the system. The second one clearly has more parallelism, but since solving each equation in the system cannot be done independently, this might not be the best approach. This motivates to investigate how they both perform. Therefore, we have formulated two parallel approaches each with their pros and cons. Thus the disadvantage of one approach is the advantage of the other. The two approaches are referred to as `S1` and `S2`.

> `S1` uses one thread to build and solve the penta/tri-diagonal system (one thread per line in the grid).
>
> `S2` uses one block to build and solve one penta/tri-diagonal system. Each block contains as many threads as there are rows in the grid (one thread for each cell in the grid).

Thus in the `S1` method each thread does a lot more work compared to `S2` , where each thread only calculate a single element in the grid and thereby more parallelism is added. Throughout the report the described convention will be used when referring to the different versions. For instance; `S12` stands for 2nd version of solution approach `S1` and `SetXMOS23` stands for 3th version of calculating x-momentum in solution approach `S2`.

Both approaches will map all data to linear memory and use their `threadIdx.x` to index what elements should be loaded. Hence the x-sweep in `S1` will index the rows in the grid by

```
k = threadIdx.x + blockIdx.x * blockDim.x + 1;
```

and run though the columns in a `for`-loop. `S2` on the other hand will index the rows and columns by

```
k = blockIdx.x + 1;      // row
j = threadIdx.x + 1;     // column
```

Notice that all arrays needed for the applications will be stored and kept on the GPU to reduce the amount of the expensive memory transfers between the host and device.

7.2 Optimization Strategy

For each parallel approach we have a naive implementation. This is done simply to have an initial version that calculates the correct solution. Furthermore it is used to investigate bottlenecks, such that these can be optimize and thereby utilize the available resources. The optimization is done in steps following the prioritized list in Section 3.13.

Prior to the optimization steps, we will use the NVIDIA Visual Profiler v4.1 tool to identify current performance bottlenecks. The profiler provide metrics and events, that will be used to investigate how the different kernels utilize the hardware and how they perform compared to each other. In this manner we can verify that a optimization step has achieved what was desired.

It shall be mentioned, that the NVIDIA Visual Profiler can report some impracticable memory throughputs. The profiler counts all transfers of the application, i.e., including all instructions and resulting overhead in performing useful operations and not transfer requested by the kernel, which are the minimum and necessary data transfer dictated by the algorithms. Therefore, we calculate our own effective bandwidth using (3.12.3) and compare this with the theoretical bandwidth to get a measurement of utilization of the available resources on the GPU.

7.2.1 Compute vs. memory bound

When optimizing is it always good to have some knowledge about which performance bounds the application have. This also makes it easier to measure how well a kernel performs on the GPU. There is usually distinguished between a compute or memory bound kernel. A compute bounded kernel is limited by the arithmetic and the instructions that has to be performed and a memory bounded kernel, by the memory that has to be transferred and therefore the memory bandwidth of the GPU.

To identify whether the application is compute or memory bound will we first investigate the source code. Since the same calculations is conducted in `xBuild` and `yBuild` and likewise in `xSolve` and `ySolve` are these for know considered to be identical. The build device function comprise of good combination of many memory load/stores and heavy arithmetic instruction, especially there are some very expensive function calls like power and square root functions and some `if`-statements. The solve device function comprise of nearly no arithmetic instruction, but a lot of memory load/stores. This indicate that building of the system is both memory and compute bounded. In fact, there are some places where there are poor memory/math overlap, which indicate that latency can affect performance. In contrast, solving the system seems to be memory bounded, and therefore limited by the memory bandwidth.

However, one could also determine the instruction per byte ratio to identify, whether the two kernels are compute or memory bounded. The perfect fp32 instructions per byte ratio is defined by the theoretical specifications (see Section 7.3), thus the used GPU, NVIDIA GeForce GTX 590, have $\sim 1.22:1$ with ECC off. By using NVIDIA Visual Profiler we profile the two kernels to get the desired events in Table 7.1

Method	Build	Solve
Instruction issued	312,426,738	26,797,120
Global store transaction	1,580,540	9,958,464
L1 global load miss	135,874,672	4,586,088

Table 7.1: Profiling the naive application `S1` in single-precision to calculate the instruction to byte ratio.

Using (3.12.1), we obtain for building the system $\sim 0.57:1$ and for solving the system $\sim 0.46:1$. Hence both kernels are memory bounded. As expected, solving the system is more memory bounded than building the system. Nevertheless, one shall keep in mind, that the source code for building the system use some very heavyweight function and have a poor memory/math overlap.

Consequently, in order to measure the effectiveness of the application, is the appropriate performance metric the effective memory bandwidth to investigate how well the kernels utilize the GPU.

7.3 Test Environment

The optimization and test of the parallel CUDA C applications are performed on the CUDA computing graphic architecture *Fermi*, with compute capability 2.0. The GPU is an NVIDIA GeForce GTX 590 (2x GF110 chip), with 1,536 MB of GDDR5 memory connected width 384-bit interface. The GPU is connected to the host via PCI-E 2.0 x16. The host comprising an Intel Xeon E5620 (12 MB cache, 2.40 GHz) CPU, with 12 GB memory and a maximum memory bandwidth of 25.6 GB/s. The serial C implementation is executed on one core.

The test system runs Ubuntu 10.04.4 LTS and the CUDA driver/runtime version is 4.1. For further details on the used system see Appendix B page 131.

The theoretical peak memory bandwidth of the device is 163.87 GB/s, since we have from (3.12.2)

$$\text{Theoretical bandwidth} = \frac{1707 \cdot 10^6 \cdot (384/8) \cdot 2}{10^9} = 163.87 \text{ GB/s}$$

The theoretical performance is 1244.15 Gflops in single-precision. NVIDIA have not published how GeForce GTX 590 perform in double-precision. Therefore,

we make use of the benchmark result from [3], which have tested the graphic card together with others for general purpose processing in both single- and double-precision, to find how the floating-point performance scales with single- and double-precision. We see that it scales with $\sim$ 6.2x. To verify the scaling for the peak computation rate in single- and double-precision, is a simple source code used, which executes a large number of multiply-add operations (FMA), (the source code is attached in Appendix E). The results are shown in Table 7.2 for two different GPUs.

GPU	Single-precision	Double-precision	Ratio
GeForce GTX 590	900.79 GFlops	141.46 GFlops	6.4x
Tesla C2070	858.53 GFlops	407.54 GFlops	2.1x

Table 7.2: Measurement the ratio of peak performance between single- and double-precision on GeForce GTX 590 and Tesla C2070.

We see that we nearly obtain the same scaling of performance from single- to double-precision as [3]. The reason why we execute the test on the Tesla C2070 is to verify our test. NVIDIA provide Tesla C2070 peak floating point performance in both single- and double-precision; cf. NVIDIA [25], it perform 1030 GFlops in single-precision and 515 GFlops in double-precision. Hence it scales with 2x, as verified by our small test.

This means that the theoretical performance for double-precision is $1244.2/6.2 = 200.68$ GFlops, which will be used to compare with the achieved Gflops in the applications.

7.3.1 Test configurations

All experiments are performed for the same set of initial test cases and same parameters: $\Delta x = \Delta y = 100$ $[m]$, $\Delta t = 20$ $[s]$, the acceleration due to gravity $g = 9.81$ $[m/s^2]$ and the Manning coefficient $M = 32$ which are unit less. Furthermore, are all representations of arrays, performed computations and used math functions performed in double-precision. This and the used parameters are desired by DHI.

For method `S1` are all tests performed on a 2048×2048 system with 64 threads per block, thus there are 2 blocks on each multiprocessor. For method `S2` are all tests performed on a 256×256 system with 256 threads per block, thus there are 16 blocks available for each multiprocessor. In both test systems is the simulations run over 100 time steps. These execution configurations are chosen such that a minimum blocks on each multiprocessor on the GPU to, i.a., insure as robust result from NVIDIA Visual Profiler as possible.

Notice that the outlined system size is the number of inner grid points in x- and y-direction. Thus each line in the system is 2 grid points larger due to keeping additional cells to impose boundary conditions. We have chosen that

the inner grid points shall be a multiple of 32, such that the number of threads is a multiple of 32. However, this can in some cases give problems with load and store effectiveness of global memory.

7.3.2 Time measuring

In all experiments CUDA event API has been used to measure the execution time. The two implemented functions `startTiming()` and `stopTiming()` are attached in Section 4.2 in the Source Code Booklet.

CUDA events has a resolution of approximately half a microsecond. The timings are measured on the GPU clock, such that the resolution is operation system independent [32, sec. 5.1.2].

7.3.3 Execution safety

Throughout the application are checks for CUDA errors performed to catch errors close to the possible error in the source code. All CUDA Runtime API returns an error code of type `cudaError_t;`, thus error handling is perform on every CUDA API and kernel calls. One can use `cutil.h`, but this do not follow with CUDA Toolkit. Therefore, we have implemented the functions `checkCudaErrors` and `getLastCudaError`, which will output a proper CUDA error string. The two implemented functions are attached in Section 4.2 in the Source Code Booklet.

7.3.4 Validation

The correctness of the serial C implementation has been validate by comparison to MIKE 21 HD. Therefore, the C implementation will be used to validate the results returned from the GPU. All elements in both water surface elevation, ζ, and flux density in both direction, p and q, will be compared to the results of the C implementation. This is done by the $\|\cdot\|_\infty$ norm of the error vector; $e = a - \hat{a}$, where a is the obtained solution from the GPU and $\hat{a}$ is the solution from the CPU [16, sec. A.3],

$$\|e\|_\infty = \max_{1 \leq i \leq n} |e_i|$$

where n is number of elements in the grid. If the difference between the results is less than $\epsilon = 10^{-12}$ will we consider them as identical and consequently we know that the parallel CUDA C implementation is correct. Thus we know that every component of the error vector between the parallel CUDA and the sequential C application cannot be greater than the $\|\cdot\|_\infty$.

Chapter 8

CUDA C Optimization

In this chapter will different massive parallel CUDA C implementation of MIKE 21 HD be described. The optimization will be an iterative process of implementing the application, investigating bottlenecks, optimize the applications and evaluating the results. The performed optimization steps will be documented by using theory of parallel programming on GPGPU, especially in CUDA, and by using the tools provided by NVIDIA. Optimization of method 1 and 2 is given in Section 8.2 and 8.3, respectively. Additional, a brief study on the performance of data transfer between host and device will be given in Section 8.1. All tests are performed with the test configurations described in 7.3.1 and double-precision is used for all applications.

8.1 Data Transfer Between Host and Device

Before we start to optimize the application will the cost of data transfer from the host to device and back again be briefly investigated. This is because if the time of data transfer is significant compared to the execution time for the application it might be useless to implement a parallel solution scheme to the GPGPU.

As described in Section 3.1 we know that host/device transfers are expensive, thus minimizing data transfers and to keep transferred data on the device is important. For this reason will all needed arrays, when simulation, be transferred and stored on the GPU. In Table 8.1, the data transfer of all needed arrays from host to device and back again is shown, respectively, for pageable and pinned memory host allocation.
For paged memory allocation the PCI-E 2.0 x16 bus speed is utilized, since we

	Method	HtoD	DtoH
Paged	Duration (ms)	41.968	51.149
	Avg. throughput (GB/s)	5.22	1.84
Pinned	Duration (ms)	38.68	15.417
	Avg. throughput (GB/s)	5.66	6.09

Table 8.1: Runtime for the combined data transfer for host-device and devise-host for respectively pageable and pinned memory allocation.

peak with same bandwidth as `bandwidthTest`[1], but device to host only utilized 44%. For pinned memory allocation we utilize the effective bandwidth for both host to device and device to host, cf. `bandwidthTest`. In fact, we obtain a 3.31x better bandwidth for the device to host transfer only by using pinned memory instead of paged memory allocation. Thus we exploit the available resources for data transfer between host and device. However, the application is not limited by the PCI-E bus speed, because at a normal simulation the number of time step will be 100-100,000. Thus the duration time for data transfer between host and device is insignificant compared to the application time, see Figure 8.1. Thus we know with experience from the following sections that data transfer between host and device will not be an important bottleneck for the application, at least as long only one GPU can be utilized.

[1] `bandwidthTest` is provided by NVIDIA CUDA C/C++ SDK Code Samples, see Appendix D on page 135.

8.2 Method 1

All source code used for method 1 can be found in Section 2 in the Source Code Booklet.

8.2.1 Naive

In the first naive approach is mainly the correctness of the program in focus. However, we insure that access in global memory, when we load and store the 10 vectors; `ama`, `amo`, `bma`, `bmo`, `cma`, `cmo`, `dma`, `dmo`, `lmo` and `rmo` is performed coalesced to optimize memory bandwidth.

The execution time for an x- and y-sweep is illustrated in Figure 8.1

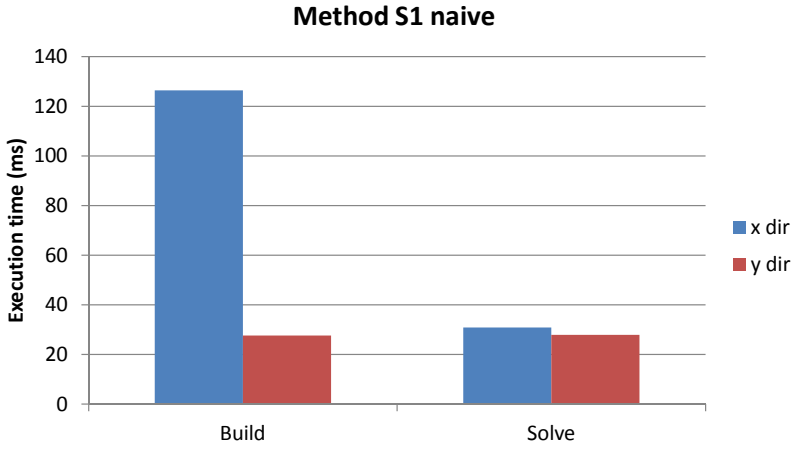


Figure 8.1: Execution time overview for method `S1N`.

The total GPU runtime is clearly dominated by the `xBuild` kernel, in fact the kernel is 4.3x slower than `yBuild` although same calculation is performed. The is also `xSolve` slower than the `ySolve`. Already after the naive implementation we have 18.6x speedup compared to the C implementation over 100 time steps (see Table 8.4).

In the following will the needed memory as function of problem size for `S1N` be determine. Assume the same number of grid points in x- and y-direction and let n be the number of inner grid points, thus we have

$$3(n+2)^2 \tag{8.2.1a}$$
$$4(n+2)(n+1) \tag{8.2.1b}$$
$$10n(n+2)+n(2n+3) \tag{8.2.1c}$$

The naive approach make use of two surface elevation for different time and a bathymetry (8.2.1a), four flux densities is needed for two different time in both x- and y-direction (8.2.1b). Further are the 10 vectors for the mass and momentum equations and the solution vector, `x`, needed (8.2.1c). Hence

$$N_{\texttt{S1N}} = 3(n+2)^2 + 4(n+2)(n+1) + 10n(n+2) + n(2n+3) \tag{8.2.2}$$
$$= 19n^2 + 47n + 20 \tag{8.2.3}$$

Thus the memory use scale asymptotically with n^2 under the above assumption. The maximum number of elements that are achievable for the naive approach on the NVIDIA GTX 590 with 1,536 MB of total mount of global memory is then given by

$$\left(19n^2 + 47n + 20\right) \cdot \texttt{sizeof(type)} = 1{,}536 \text{ MB} \tag{8.2.4}$$

where `type` is ether `double` or `float` size 8 bytes and 4 bytes, respectively. Thus systems size can approximately be 3253×3253 for `double` and 4601×4601 for `float`. However, this is a theoretical value so it is not likely to achieve exactly these system sizes. Figure 9.3 illustrate how far we can come with the different approaches on the test environment.

8.2.2 Naive version 2

To investigate the performance difference in the sweeps and how the naive approach perform in general on the GPGPU we gatherers metrics/events from the NVIDIA Visual Profiler shown in Table 8.2.

Method	`xBuild`	`xSolve`	`yBuild`	`ySolve`
Duration (ms)	126.394	30.86	27.62	27.922
Requested gld throughput (GB/s)	13.35	37.44	49.76	41.38
Requested gst throughput (GB/s)	2.97	14.17	13.57	15.66
gld throughput (GB/s)	53.38	37.46	96.45	41.4
gst throughput (GB/s)	2.97	20.24	13.58	16.22
gld efficiency (%)	25	100	51.3	100
gst efficiency (%)	100	70	100	96.6
L1 gld hit rate (%)	0	5.1	80.2	5.2
IPC	0.083	0.155	0.352	0.17

Table 8.2: Metrics and events from NVIDIA Visual Profiler for the naive approach `S1N`.

We see that the global memory load efficiency is low for `xBuild` and `yBuild` . The global memory load/store efficiency shows the ratio of requested global memory load/store throughput to the actual global memory load/store throughput. Thus bandwidth is being wasted if the efficiency is not 100%. `xBuild` only

has 25% global load efficiency and `yBuild` 51.3%.Therefore, the access patterns of `xBuild` is likely scattered and in concerned with `yBuild` the access patterns could be scattered or misaligned.

The L1 global load hit rate shows that the cache is being relatively utilized for the `yBuild` kernel, especially compared to `xBuild` which do not at all make use of the fast cache. At the same time, we see that IPC i relative low and the memory throughput relative high, which indicate that the application is memory bounded as described in Section 7.2.1.

Hence x-direction have a poor performance compared to y-direction, because memory access is uncoalesced. When examining how an x- and an y-sweep accesses the elements in the grid the reason becomes immediately clear. The threads in an x-sweep load elements with a stride of `sizeX` in parallel, thus accessing to global memory are not coalesced and thereby a lot of elements in the transaction are not used, which results in wasted bandwidth. Illustration of the different access are shown in Figure 8.2

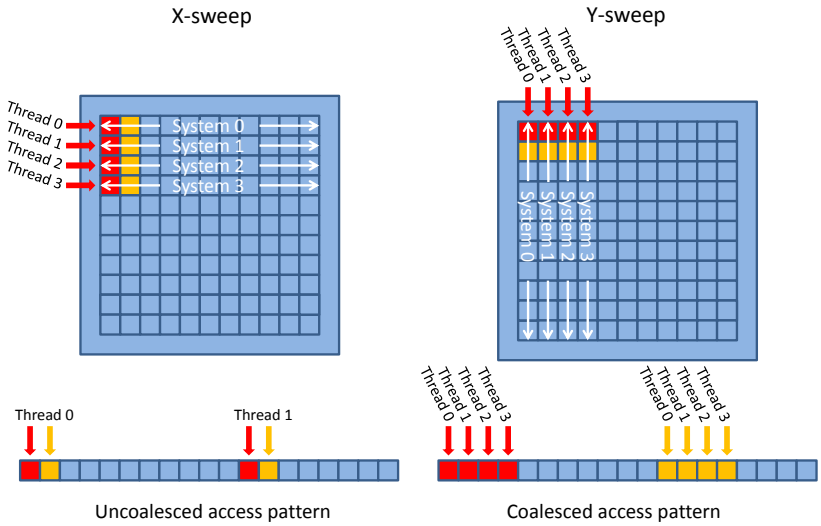


Figure 8.2: Access pattern for `S1N` for a x- and y-sweep.

In order to obtain better coalesced access to global memory will the x-sweep be modified, so we first transpose the grids then execute a normal y-sweep and then transpose again as illustrated on Figure 8.3. The transposing is performed massive parallel on the GPGPU. We make use of CUDA C/C++ SDK Code Samples[2]

[2]CUDA C/C++ SDK Code Samples follows with CUDA TOOLKIT [24].

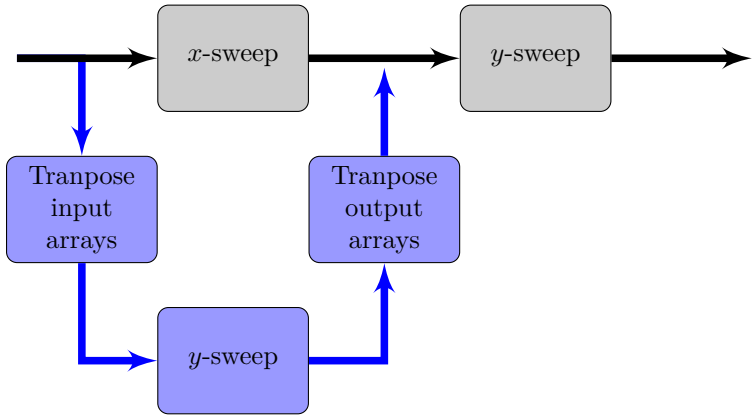


Figure 8.3: Flowchart through one simulation time step in the naive `S1` approach shown in gray and the modified x-sweep in blue.

This simple optimization step is implemented in `S1N2`. The implementation results in the execution times illustrated together with the old x-sweep in Figure 8.4.

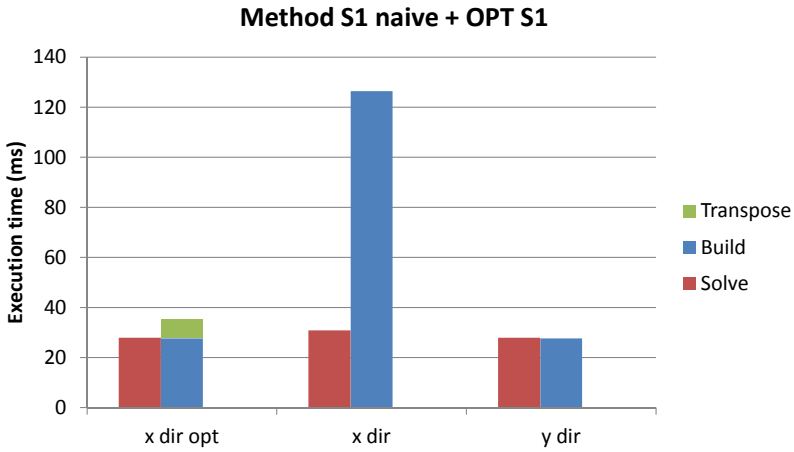


Figure 8.4: Execution time overview for `S1N` and `S1N2`.

We see that transposing the grids (in fact 7 arrays twice per sweep) and applying `yBuild` is 3.6x faster than the original `xBuild`. In Table 8.3 some gathered performance metric is shown for transposing one array of size 2048×2048.

Method	One transposing
Duration	0.534 ms
Bandwidth	120 GB/s
IPC	1.111
Achieved Occupancy	0.92
Shared Bank conflicts	0

Table 8.3: Gather performance metric for one transposing of one array with size 2048×2048.

The effective bandwidth is calculated by using equation (3.12.3), thus

$$\text{Effective} = \frac{\left((2048+2)^2 \cdot 8 \cdot 2\right)/1024^2}{0.534} = 120\text{GB/s}$$

Hence the transpose kernel obtain 73% of the theoretic bandwidth and 92% of the bandwidth achieved by `bandwidthTest` (see Appendix D on page 135). Consequently, the transpose kernel utilize the available resources pretty good.

Additional we see that the achieved occupancy is 0.92, i.e., the opportunity to hiding memory latencies is there and thereby keep the multiprocessors on the GPU busy.

Performance evaluation

Comparing the execution time of the two GPGPU versions `S1N` and `S1N2` to the CPU version, we achieve the results shown in Table 8.4.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	398.191 s	-	-	-
`S1N`	21.391 s	18.6x	23.9 GB/s	8.3 Gflops
`S1N2`	11.898 s	33.5x	43.0 GB/s	9.9 Gflops

Table 8.4: Comparing execution time of the seqential C implementation on the CPU against `S1N` and `S1N2` on a 2048×2048 system for 100 time step in double-precision.

We see that a simple transposing arrays and thereby insure to access global memory coalesced result in almost halve the execution time of the entire application and is therefore very significant.

The naive parallel implementation is already 18.6x faster than the C implementation, which indicate that using massive parallelism to solve the shallow water equations is advisable, and that we after simple optimization have a speedup on 33.5x compared to the C implementation. However, the obtained bandwidth indicate that we do not utilize the available resources.

8.2.3 Version 2

Even though `yBuild` performs a lot better than `xBuild` due to the more coalesced pattern, it still only has a 51.3% global load efficiency. This is because that even though the threads load the data aligned, each thread has to load several values that are placed right next to each other in global memory as illustrated on Figure 8.5.

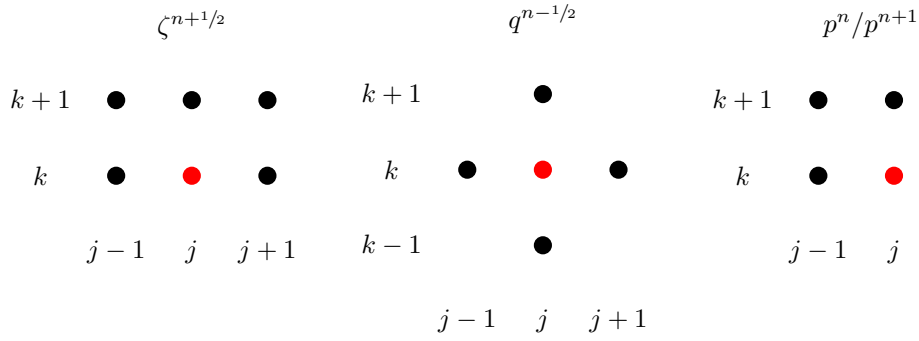


Figure 8.5: Needed values for calculation of a mass and momentum equation at (j,k) for respectively $\zeta^{n+1/2}$, $q^{n-1/2}$ p^n/p^{n+1} in an y-sweep.

We see, e.g., for $\zeta^{n+1/2}$ and $q^{n-1/2}$ that they need the values (j,k), $(j-1,k)$ and $(j+1,k)$ for place (j,k). When each thread has to access three values right next to each other in global memory they will have an access pattern similar to the ones illustrated in Figure 8.6, 8.7 and 8.8.

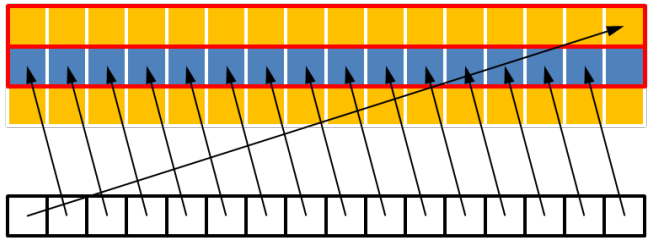


Figure 8.6: Missaligned access pattern where the first thread accesses values in a separate 128 byte segment.

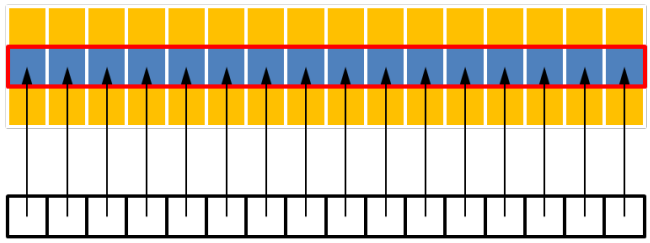


Figure 8.7: Perfectly coalesced access pattern

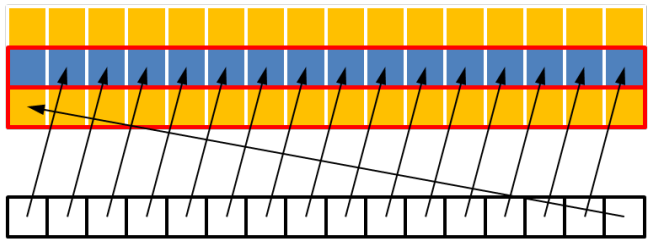


Figure 8.8: Missaligned access pattern where the last thread accesses values in a separate 128 byte segment.

The 1st and 3th misalignment access pattern results in caching two 128 byte segment, where only a single element in one of the 128 byte segment is to be used. This is of course very inefficient.

To improve this a solution could be to use non-caching loads, see Section 8.2.3, or to load the data into shared memory and then do all the reads from there. We try two different approaches of loading data into shared memory

1. Loading all values into shared memory using the entire thread block, but use two threads less to do the calculations. This is necessary since otherwise they would request values that has not been loaded into shared.
2. Loading only the "center" values $\zeta_{j,k}^{n+1/2}$, $\zeta_{j,k+1}^{n+1/2}$ and $q_{j,k}^{n-1/2}$ into shared memory. Using this approach all threads can work, but all other values have to be loaded from L1/L2. This approach might be beneficial since the other values only are read a limited amount of times and therefore does not need to be specific handled to obtain good performance.

These two approaches are implemented. Performance results from the NVIDIA Visual Profiler are shown in Table 8.5, where shared memory method *all* refer to the 1st approach described and *"center"* to the 2nd approach.

On top of that reducing overall reads and writes to global memory is always relevant. Since the vectors `ama`, `bma` and `cma` are all just defined as constant, and `cma` is the only values that ever change (which first happens in the solver)

it would be smarter to just store them as a constant and in that way reducing global memory transactions. Further there is no reason to have a separate solution vector `x`, since the values easily can be stored in `dma` and `dmo`. All this is done for both methods described above, see Section 2.6 in the Source Code Booklet.

We get the following performance specifications from the NVIDIA Visual Profiler.

Method Shared memory method	`yBuild` all	`yBuild` "center"	`ySolve`
Duration (ms)	28.171	25.807	21.379
Requested gld throughput (GB/s)	24.58	36.31	39.44
Requested gst throughput (GB/s)	9.98	10.89	20.45
gld throughput (GB/s)	48.81	68.11	39.43
gst throughput (GB/s)	11.25	10.89	21.18
gld efficiency	50.1%	52.7%	100%
gst efficiency	88.7%	100%	96.6%
L1 gld hit rate	62.3%	74.6%	3.7%
IPC	0.355	0.371	0.128

Table 8.5: Metrics/Events from NVIDIA Visual Profiler for `S12`.

It is clear that the two approaches are very similar in performance, but the approach where only the "central" values are loaded into shared memory does outperform the other. In fact, loading all values into shared memory decreases performance compared to the naive `yBuild`. This is most likely because the GPU are able to use the L1 cache more efficiently than our shared memory approach. For this reason we will focus on the "central" value approach.

The biggest performance boost is actually in the solver due to the reduced global memory access. Even though most memory throughputs falls including the L1 global load hit rate, the time goes from 27.922 ms for the naive solver to 21.379 ms which gives 1.3x speedup.

Overall the new kernels gives 38.7x speedup over 100 time steps compared to the C implementation.

L1 cache

From the results in Table 8.5 it is clear that L1 cache has an influence on the performance, since loading more values to shared memory actually made the performance worse. It is therefore interesting to investigate the influence and optimal settings for the L1 cache.

The `yBuild` still has a pretty low global load efficiency. This could be improved by non-caching in L1 cache, such that smaller segments sizes can be transferred and thereby better utilization of the available bandwidth can be

obtained. Also since the `ySolve` only has a 3.7% L1 global load hit rate it might be more efficient to not even try to hit in L1, but always load directly from L2. To investigate this we use the compiler flag `-Xptxas -dlcm=cg` to turn off the L1 cache and thereby only cache in L2. The results can be seen in Table 8.6

Method	`yBuild`	`ySolve`
Duration (ms)	28.611	21.374
gld efficiency	85.1%	100%
gst efficiency	100%	96.6%

Table 8.6: Metrics from NVIDIA Visual Profiler for `S12` when caching and non-caching in L1.

As expected, we see that the global load efficiency has increased significantly for `yBuild`. This is because when a warp requests a single element, that before would have been answered with a 128 byte segment, now can be answered with a 32 byte segment and thereby increasing efficiency (utilize bus). However, this does unfortunately not decrease the running time of `yBuild`. In fact, it increases with 11%! So even though more unnecessary data is transferred the higher bandwidth to the L1 cache yields a better performance.

When looking at `ySolve` it is clear that non-caching in L1 has changed absolutely nothing. This is not that surprising, since the load and store efficiency already was very high and could not be improved much.

Overall it is clear that caching in L1 does improve the performance so maybe increasing L1 cache size gives even better performance. In the application we do not use that much shared memory, it is therefore also interesting to investigate what impact it will have to increase the size of L1 cache from 16 kB to 48 kB. This can be done by adding the function calls `cudaFuncSetCacheConfig(yBuildS12,cudaFuncCachePreferL1);` and `cudaFuncSetCacheConfig(ySolveS12,cudaFuncCachePreferL1);` into the host code. The results are shown in Table 8.7

Again we see that L1 has a positive effect on `yBuild`. It decreases the runtime by 7%, gives a better load throughput and gives a higher L1 hit rate. However since we still do not reuse a lot of element in the solver the impact here is insignificant.

Overall we find that L1 has a substantial impact on the performance of `yBuild`. The execution time goes from 28.611 ms when non-caching in L1 to 24.043 ms when fully utilizing the possible L1 cache size. A performance gap of 19%!

Method	yBuild	ySolve
Duration (ms)	24.043	21.339
Requested gld throughput (GB/s)	38.98	39.51
Requested gst throughput (GB/s)	11.69	20.49
gld throughput (GB/s)	73.95	39.52
gst throughput (GB/s)	11.64	21.22
gld efficiency	52.7%	100%
gst efficiency	100%	96.6%
L1 gld hit rate	86.4%	3.8%

Table 8.7: Metrics from NVIDIA Visual Profiler for `S12` with 48 kB L1 cache size.

Performance evaluation

Comparing the execution time of the GPGPU version `S12` to the CPU version, we achieve the results shown in Table 8.8.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	398.191 s	-	-	-
`S12`	9.870 s	40.3x	51.9 GB/s	17.9 Gflops

Table 8.8: Comparing execution time of the CPU against `S12` on a 2048×2048 system for 100 time step in double-precision.

We see that `S12` obtain the same solution 40.3x faster than the C implementation. Further that we utilize 32% of the theoretic bandwidth. It is not the best, but acceptable for now.

8.2.4 Version 3

The memory usage has now been improved to a satisfiable level and it is time to optimize the instructions. Again we use the NVIDIA Visual Profiler to discover potential performance bottlenecks. The results can be seen in Table 8.9 where divergent branches, control flow divergence and replayed instruction are calculated based on the obtained values.

Method	`yBuildS12`	`ySolveS12`
Duration (ms)	24.043	21.339
Inst. issued	99,378,225	26,732,961
Inst. executed	92,673,121	26,470,336
Threads inst. exec.	2,964,085,109	847,050,752
Branch	6,159,781	1,572,864
Div. branch	8,214	0
IPC	0.396	0.128
Shared bank conflict	0	0
Ratio Div. branches	0.13%	0%
Control flow div.	0.05%	0%
Replayed inst.	6.75%	0.98%

Table 8.9: Events from NVIDIA Visual Profiler for `S12`.

Overall it is clear that there are no big instruction problems with the programs. Both kernels show very little divergent branches, control flow divergence and replayed instructions. The approach is therefore not to try to improve any of those but simply to bring down overall issued instructions and simplify calculations.

Simplify calculation of momentum equation `bmo`

First we investigate the device function `SetYMO`, where the calculation of `bmo` in particular seems complicated. The code is shown in Listing 8.1.

Listing 8.1: Calculation of `bmo`

```
1 hr0 = 1.0/hjk[thid];
2 hr1 = 1.0/hjkp1[thid];
3 hres = sqrt(2.0/(hr0*hr0+hr1*hr1));
4 C = M * pow(hres,1.0/6);
5 ...
6 bmo[idx] = ddt + (g * sqrt(qjk[thid]*qjk[thid]+pst*pst))/(C*C*hres*hres
      ) - cvl + cvr;
```

So alone to calculate the denominator in `bmo` a square root operation, a power operation, five divisions, six multiplications and four different variables are used. That does not seem very effective, especially considering that power and square root are very expensive operations.

In order to improve this, we want to simplify the expression by first expanding the term to understand it better

$$
\begin{aligned}
&\frac{1}{C \cdot C \cdot hres \cdot hres} \\
&= \frac{1}{\left(M \cdot hres^{1/6}\right)^2 \cdot hres^2} \\
&= \frac{1}{\left(M \cdot \sqrt{\frac{2}{hr_0^2 + hr_1^2}}^{1/6}\right)^2 \cdot \left(\sqrt{\frac{2}{hr_0^2 + hr_1^2}}\right)^2} \\
&= \frac{1}{\left(M \cdot \left(\sqrt{\frac{2}{\left(\frac{1}{h_{j,k}}\right)^2 + \left(\frac{1}{h_{j,k+1}}\right)^2}}\right)^{1/6}\right)^2 \cdot \left(\sqrt{\frac{2}{\left(\frac{1}{h_{j,k}}\right)^2 + \left(\frac{1}{h_{j,k+1}}\right)^2}}\right)^2}
\end{aligned}
$$

This clarify, that the calculation is quite complex, but it is also obvious that some of the instructions, as for instance the square roots, are unnecessary. We will now reduce the complexity a bit.

$$
\begin{aligned}
&\frac{1}{M^2 \cdot \left(\frac{2}{\frac{1}{h_{j,k}^2} + \frac{1}{h_{j,k+1}^2}}\right)^{1/6} \cdot \left(\frac{2}{\frac{1}{h_{j,k}^2} + \frac{1}{h_{j,k+1}^2}}\right)} \\
&= \frac{1}{M^2} \cdot \left(\frac{1}{2 \cdot h_{j,k}^2} + \frac{1}{2 \cdot h_{j,k+1}^2}\right)^{1/6} \cdot \left(\frac{1}{2 \cdot h_{j,k}^2} + \frac{1}{2 \cdot h_{j,k+1}^2}\right) \\
&= \frac{1}{M^2} \cdot \left(\frac{h_{j,k}^2 + h_{j,k+1}^2}{2 \cdot h_{j,k}^2 \cdot h_{j,k+1}^2}\right)^{1/6} \cdot \frac{h_{j,k}^2 + h_{j,k+1}^2}{2 \cdot h_{j,k}^2 \cdot h_{j,k+1}^2}
\end{aligned}
$$

Thus we end up with a much simpler expression and since `M` is just a constant, we can pre-calculate $\frac{1}{M \cdot M}$. We then have the code shown in Listing 8.2

Listing 8.2: Improved calculation of `bmo`

```
1 hr0  = hjk[thid]*hjk[thid];
2 hres = hjkp1[thid]*hjkp1[thid];
3 hres = (hres+hr0)/(2*hres*hr0);
4 hres *= pow(hres,1.0/6) * M * M;
5 ...
6 bmo[idx] = ddt + (g * sqrt(qjk[thid]*qjk[thid]+pst*pst))* hres - cvl +
      cvr;
```

This simplification results in only a power operation, two divisions, 7 multiplications and 2 different variables. Hence the calculations are much simpler and less expensive.

Removing `if`-statements

In the device function `SetYMO` are `if`-statements with exactly the same conditions used when defining `f1`, `f2` and `dmo`. Thus it is unnecessary to have them all. An example of the `if`-statements can be seen in Listing 6.2. On top of that the small number of divergent branches that exists for `yBuild` properly originate from these `if`-statements. So both the number of instructions issued and the divergent branches can potentially be brought down by having only one simpler `if`-statement.

The reduction is done by having an initial `if`-statement where we define a few binary variables and use those to control the calculations instead, see Listing 8.3. In this manner we avoid scenarios where threads have to load different values depending on which branch they take in the `if`-statement.

Listing 8.3: Improved control flow; branching and divergence.

```
1  if (bL[jm1+nY]==lh)
2  {
3      bin1 = 0;
4      bin2 = 1;
5      ...
6  }
7  else if (bL[jm1+kp1]==lh)
8  {
9      bin1 = 1;
10     bin2 = 0;
11     ...
12 }
13 else
14 {
15     bin1 = 1;
16     bin2 = 1;
17     ...
18 }
19
20 f1 = (pNp1[jm1+nP] + pNp1[jm1+pkp1]);
21 f1 /= (hjk[thid]*(2-bin1) + hjkp1[thid]*(2-bin2) + (zetaNp05[jm1+nY] -
       bL[jm1+nY])*bin1 + (zetaNp05[jm1+kp1] - bL[jm1+kp1])*bin2);
```

Hereby are line 21 calculated differently depending on the outcome of the `if`-condition.

Optimizing `LcElim`

When investigating the device function `LcElim` in `ySolve` we also find that improvements can be made. Recall that the function eliminates the values in

the penta-diagonal to a tri-diagonal matrix system by two `for`-loops. The first `for`-loop eliminates the values in the sub-diagonal and the second eliminates the super-diagonal.

The system matrix is a penta-diagonal matrix because of the values in the vectors `lmo` and `rmo` from the momentum equation. Since in the penta-diagonals there are only value for every other elements (momentum equations) these are eliminated using the mass equations. This means that all the calculations in the loops are independent of each other. Therefore, there is no reason to use two `for`-loops. Instead we implement a `for`-loop that eliminates both diagonals simultaneously. Thereby, less instructions are issued and less global writes are performed, since we only update each element once.

It is done by having an initial step then the optimized `for`-loop and at last a final step as shown in Listing 8.4.

Listing 8.4: Optimized `LcElim`.

```
__device__ void
LcElimS13(Td dydt, Td *dma, ...)
{
    int i, ii, s, e;
    Td l, r;

    s = 1-bcs;
    e = sizeX+bce-1;

    ii = s*sizeX+idx;
    r = rmo[ii];
    bmo[ii] += r;
    cmo[ii] -= dydt*r;
    dmo[ii] -= dma[ii+sizeX]*r;

    for (i=s+1; i<e-1; i++)
    {
        ii += sizeX;

        l = -lmo[ii];
        amo[ii] -= dydt*l;

        r = rmo[ii];
        cmo[ii] -= dydt*r;
        bmo[ii] += r-l;
        dmo[ii] -= dma[ii]*l
                +dma[ii+sizeX]*r;
    }
    ii += sizeX;
    l = -lmo[ii];
    amo[ii] -= dydt*l;
    bmo[ii] -= l;
    dmo[ii] -= dma[ii]*l;
}
```

Listing 8.5: Original `LcElim`.

```
__device__ void
LcElimS12(Td dydt, Td *dma, ...)
{
    int i, ii, s, e;
    Td dd = 0.0f;

    s = 1-bcs;
    e = sizeX+bce-1;

    ii=s*sizeX+idx;
    for (i=s; i<e; i++)
    {
        ii += sizeX;
        if (lmo[ii] == 0.0f)
            { continue; }

        dd = -lmo[ii];
        amo[ii] -= dydt*dd;
        bmo[ii] -= dd;
        dmo[ii] -= dma[ii]*dd;
    }
    for (i=s; i<e; i++)
    {
        ii-=sizeX;
        if (rmo[ii] == 0.0f)
            { continue; }

        dd = rmo[ii];
        bmo[ii] += dd;
        cmo[ii] -= dydt*dd;
        dmo[ii] -= dma[ii+sizeX]
                    *dd;
    }
}
```

Optimizing Thomas algorithm

As mentioned the Thomas algorithm consists of a forward elimination and a backward substitution phase. As implemented now the backward substitution phase runs a `for`-loop, where the solution is obtained and stored in `dma` and `dmo`. Immediately after are the solution from `dma` and `dmo` written into `zetaNp1` and `qNp05`, respectively, also in a `for`-loop. There is no good reason why the solution should be written temporarily to `dma` and `dmo` first. Therefore, we modify the backward substitution phase so that it stores the elements directly into `zetaNp1` and `qNp05` as shown in Listing 8.6.

Listing 8.6: Improved backward substitution from `ySolveS13`.

```
1  double dma1, dmo1;
2  idx = threadIdx.x + blockIdx.x * blockDim.x;
3
4  ...
5
6  // Store the determined solutions
7  jk = j+sizeY*sizeGx;
8  idx += sizeY*(sizeY);
9
10 dma1 = dma[idx];
11 zetaNp1[jk] = dma1;
12
13 for(i=1; i<sizeY; i++)
14 {
15     jk -= sizeGx;
16     idx -= sizeY;
17
18     dmo1 = dmo[idx]-cmo[idx]*dma1;
19     qNp05[jk]   = dmo1;
20
21     dma1 = dma[idx]-cma[idx]*dmo1;
22     zetaNp1[jk] = dma1;
23 }
```

The solution is temporarily stored in the register variables `dma1` and `dmo1` to reduce even more global loads. In this way a `for`-loop less have to be executed, which means a reduction in instructions (counter overhead) and in global data transfers.

Performance evaluation

By the optimization steps described above, we obtain the values shown in Table 8.10 from the profiler.

Method	yBuildS12	ySolveS12	yBuildS13	ySolveS13
Duration (ms)	24.043	21.339	17.953	14.576
Inst. issued	99,378,225	26,732,961	86,881,283	20,967,449
Inst. executed	92,673,121	26,470,336	80,840,198	20,704,896
Threads inst.exec.	2,964,085,109	847,050,752	2,586,837,366	662,556,672
Branch	6,159,781	1,572,864	3,275,542	1,048,896
No. div. branch	8,214	0	0	0
IPC	0.396	0.128	0.465	0.147
Bank conflicts	0	0	0	0
Ratio div. branch	0.13%	0%	0%	0%
Control flow div.	0.05%	0%	0%	1.25%
Replayed inst.	6.75%	0.98%	6.98%	1.25%

Table 8.10: Events from NVIDIA Visual Profiler for `S12` and `S13`.

The performed optimization steps had a significant influence on the performance of the kernels. Comparing the execution time for building the system, we see that `yBuild13` obtain the same 1.3x faster than `yBuild12` and as expected, we see that the ratio of divergent branches has dropped to zero due to the simplification of the `if`-statements. The percentage of replayed instructions seems to have increased a bit, but this is only due to the 13% fall in issued instructions. The solver perform 1.5x speedup compared to `ySolve12` execution time. This is of course partly due to the 22% fewer instructions issued, but also due to the fewer global writes that followed by the optimization.

Performance evaluation

Comparing the execution time of the GPGPU version `S13` to the CPU version, we achieve the results shown in Table 8.11.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	398.191 s	-	-	-
`S13`	7.325 s	54.4x	69.9 GB/s	24.2 Gflops

Table 8.11: Comparing execution time of the CPU against `S13` on a 2048×2048 system for 100 time step.

We see that `S13` obtain the same solution 54.4x faster than the C implementation. Further that we utilize 43% of the theoretic bandwidth.

8.2.5 Version 4

We have now performed the most crucial performance steps. In this section will different kinds of optimization steps be performed in order to utilize the available resources as must as possible .

Occupancy

In the test system with 2048×2048 grid points we have 64 threads per block, thus there are 2 blocks on each SM. From [29, Table F-2] we know that Compute Capability of 2.0 has a maximum of 1536 resident threads per SM. Hence from (3.9.1) we have that

$$\text{Occupancy} = \frac{2 \cdot 64}{1536} = 0.083$$

This is likely to be a performance bottleneck, since it will be hard to hide memory and instruction latencies and keep the GPU busy using multi-threading. However, in order to increase the resident threads per SM is it necessary to increase the system size, because of the chosen solution method. This means with this method occupancy can not be improved without changing the system size. There is also a limit to the possible system size due to the DRAM on the GTX 590 and therefore we are limited on the size of possible simulations, as described in Section 8.2.5.1.

Using CUDA *Occupancy Calculator*[3], where we, i.a., can see the trade-offs between thread count and register use. Compiling `S13` with the compiler option `--ptxas-options=-v` to `nvcc` we achieve Listing 8.7

Listing 8.7: Register count for `S13`

```
1 ptxas info    : Compiling entry function '
      _Z9yBuildS13PdS_S_S_S_S_S_S_S_S_S_S_iiiiddddddddd' for 'sm_20'
2 ptxas info    : Function properties for
      _Z9yBuildS13PdS_S_S_S_S_S_S_S_S_S_S_iiiiddddddddd
3     16 bytes stack frame, 16 bytes spill stores, 24 bytes spill loads
4 ptxas info    : Used 63 registers, 1536+0 bytes smem, 224 bytes cmem
      [0], 16 bytes cmem[14], 48 bytes cmem[16]
5 ptxas info    : Compiling entry function '
      _Z9ySolveS13PdS_S_S_S_S_S_S_S_iiid' for 'sm_20'
6 ptxas info    : Function properties for
      _Z9ySolveS13PdS_S_S_S_S_S_S_S_iiid
7     0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
8 ptxas info    : Used 38 registers, 136 bytes cmem[0], 16 bytes cmem
      [14], 32 bytes cmem[16]
```

Here we see that `yBuild` use 63 registers and 1536 bytes shared memory. Paste this values into CUDA *Occupancy Calculator* we achieve Figure 8.9a illustrating the impact of varying block size and Figure 8.9b illustrating the impact of varying register count per thread.

[3] Provided tool in CUDA Toolkit [27].

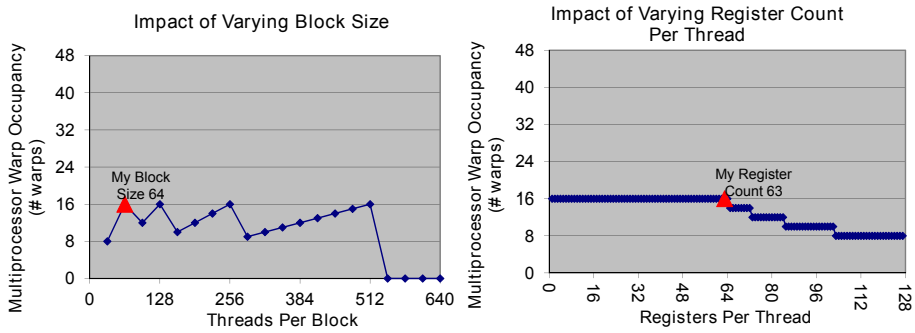


Figure 8.9: Impact on occupancy when varying block size 8.9a and registers per thread 8.9b for `S13`.

The theoretical occupancy of each multiprocessor is 33%, since the total number of 32-bit registers per multiprocessor is 32,768 for Compute Capability 2.0 [29, Table F-2], this imply that there maximum can be 8 blocks per multiprocessor. Hence we have from (3.9.1)

$$\text{Occupancy} = \frac{8 \cdot 64}{1536} = 0.333$$

However, decreasing register count will not change the warp occupancy (see Figure 8.9b), because we are limited by not having enough threads. As mentioned increasing the number of threads will decrease the number of blocks per multiprocessor with this solution method, `S1`, since we are limited by the size of DRAM on the GPU. Therefore the theoretical occupancy is not obtainable.

Register spilling

In Listing 8.7, we see that kernel `yBuild` use 63 register there is register spilling register spilling. We will therefore investigate whether local memory usage has an impact on performance for memory and/or instructions. Profiling `S13`, particularly `yBuild` kernel, we obtain the counters listed in Table 8.12

Method	yBuild13
L1 local load hit	1,168,552
L1 local load miss	10,720
L1 local store hit	200
L1 local store miss	393,280
instruction issued	86,881,283
L2 query read	7,007,489
L2 query write	7,337,988
gld request	3,013,960
gst request	917,120

Table 8.12: Profile counters for yBuild in S13.

The L1 local load hit is 99.09%, which indicate that L1 contains most of the spills.

Impact on memory Estimated L2 queries of all 16 multiprocessor due to local memory is $2 \cdot 4 \cdot \#\text{SM} \cdot \text{L1 local load miss} = 1{,}372{,}160$. We multiplied with 2 because a load miss implies that a store happened first and with 4 because a local memory transaction is 128 byte which is equal to 4 L2 transactions. The percentage of all L2 queries due to local memory is $1{,}372{,}160/(7{,}007{,}489 + 7{,}337{,}988) = 0.096$. In other words only 9.6% of memory traffic between the multiprocessor and L2/DRAM is due to local memory.

The impact on memory throughput is investigated by comparing L1 local load miss count to global load and store memory count: (gld request + gst request)/(2·L1 local load miss). We see that hardly any bus traffic to global memory is due to spills, since the ratio of global memory to local memory bus traffic is approximately 183 : 1.

Impact on instruction The percentage of instructions due to local memory is simply the total instructions due to local memory over the number of instructions issued, hence $(10{,}720 + 1{,}168{,}552 + 393{,}280 + 200)\,/86{,}881{,}283 = 0.018$. Thus only 1.8% of instructions are due to local memory.

Consequently, these percentages tell that register spilling does not have a significant impact on the performance. Removing spilling completely will improve performance no more than 9.6% or 1.8% if the kernel is memory- or instruction bounded respectively.

The kernel uses a lot of registers, but this does not have any significant impact on the performance and reducing it will not increase occupancy. Besides, registers is the fastest memory on the GPU and using registers will reduce other kinds of memory traffic. Additional in NVIDIA's next generation of CUDA compute architecture named *Kepler* will it be possible to have up to 255 registers per

thread! This means that the maximum of 32-bit registers per multiprocessor is 65,536, and register spilling will no longer be a problem [34] for this system.

Latency hiding

In respect to Section 8.2.5, how do we obtain better performance at lower occupancy? We do not, as mentioned, have enough threads per multiprocessor to exploit Thread Level Parallelism (TLP) and by that getting higher occupancy. Additional Instruction Level Parallelism (ILP) can be used to hide latency and by that achieve higher efficiency at lower occupancy.

Therefore, we go through the code to obtain as many independent instructions after each other as possible simple to add more ILP and by that hiding register dependencies and hiding global memory latency, see example in Section 3.8.

Change of calculation flow

We have maintained the same calculation flow as in MIKE 21 HD, see Figure 6.2. This approach imply unnecessary global memory loads and stores of values in the vectors `dma`, `amo`, `bmo`, `cmo`, `dmo`, `lmo` and `rmo`. Therefore, we remove the elimination device function `LcElim` and instead build the tri-diagonal system in the build function `yBuild`. Thereby we can remove vectors `lmo` and `rmo` and reduce global memory transaction by six loads and six stores per thread per column.

Further optimization

The last optimization step is introducing the keyword `__restrict__` in all kernel calls, such that the compiler can reorder and do common sub-expression elimination at will. Further are the compiler forced to inline device function in the kernel with the qualifier `__forceinline__`.

It has also been investigated if unrolling `for`-loops using `#pragma unroll` can improve performance by reducing counter overhead. This however turned out to be insignificant for the performance so the `#pragma unroll` is not used in the following applications.

In the code we make use of a square root and a power function. These two functions are very expensive because of the double-precision (Full IEEE 754-2008 64-bit precision). We tried to use a CUDA double-precision floating-point square root function `__dsqrt_rd(x)`, but this had almost no effect. However, to show how expensive the power function in double-precision is, will a single-precision floating-point power function `powf(x,y)` be applied together with the compiler option `−use_fast_math`. This simple switch improve the performance by 16% and 19% improvement is obtained when a single-precision floating square root function `sqrtf(x)` also is used. Be aware that the performed calculation is not in double-precision any more and thereby we get an error compared to

the double-precision CPU version on the 10^{-8} position after 100 time steps. Therefore, this will not be implemented now, since we want an exact solution compared to MIKE 21 HD. Nevertheless, this huge impact on the performance when so little changes is applied, motivate to further investigate the performance impact of calculating in single-precision against double-precision in return of less precision, see Section 9.4.

8.2.5.1 Performance evaluation

The described optimization step is implemented in `S14`. These steps have not increase register spill, in fact it has now been removed. Comparing the execution time of the GPGPU versions `S13` and `S14` to the CPU version, we achieve the results shown in Table 8.13.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	398.191 s	-	-	-
`S13`	7.325 s	54.4x	69.9 GB/s	24.2 Gflops
`S14`	6.095 s	65.3x	84.0 GB/s	29.0 Gflops

Table 8.13: Comparing execution time of the CPU against `S13` and `S14` on a 2048×2048 system for 100 time step in double-precision.

The execution time performance have improved by 16% or about 1.20x speedup compared with `S13` on a 2048×2048 with 100 time step. Thus `S14` obtain the same solution as the C implementation, but 65.3x faster! The achieved effective bandwidth is 51% of the theoretic bandwidth. The reason why the bandwidth is not that high can be found in the system size as described in Section 7.3.1. In Table 9.4 are the performance as a function of the system size illustrated.

The profiler state that the IPC for `yBuildS14` and `ySolveS14` is 0.488 and 0.182 out of 2.0, respectively, while the memory throughput is 84.0 GB/s out of 163.87 GB/s. Hence the entire application is memory bounded (limited by the hardware bandwidth), as expected. Which also can be seen of the small performed flops, thus comparing with theoretic flop is not an appropriate performance metric for this application.

Besides better performance are obtained from the naive version to `S14` have we also reduced the amount of used vectors. With (8.2.1) in mind will we in the following determine the used memory as function of problem size for `S14`. Assume the same number of grid points in x- and y-direction and let n be the number of inner grid points, thus we have

$$3(n+2)^2 \tag{8.2.5a}$$
$$4(n+2)(n+1) \tag{8.2.5b}$$
$$(n+2)^2 + (n+2)(n+1) \tag{8.2.5c}$$
$$6n(n+2) \tag{8.2.5d}$$

The different between `S1N` and `S14` is that only 6 vectors to mass and momentum equations are needed and that the solution vectors, `x`, is removed (8.2.5d). Additional, two extra vector are needed when transposing between each sweep (8.2.5c). Hence

$$N_{\texttt{S14}} = 3(n+2)^2 + 4(n+2)(n+1) + (n+2)^2 + (n+2)(n+1) + 6n(n+2) \tag{8.2.6}$$

$$= 15n^2 + 43n + 26 \tag{8.2.7}$$

Considering (8.2.2) one can approximately see a factor on 15/19. Hence `S14` can simulate systems with 21% more elements than `S1N` without using more memory. This is significant, since the memory space on graphic cards (GPU) is finite and very small compared to the host memory. This implies that on the test set up `S14` can approximately simulate systems sizes on 3661×3661 for `double` and 5179×5179 for `float`. Again, keep in mind that this is theoretic so it likely not possible to achieve exact this system size.

Assigning attention to the transpose kernel, which transpose 7 arrays twice per sweep, we see that it is responsible for 12% of the total application execution time (see performance metric for transpose kernel in Table 8.3), but this is well spent especially compared to accessing elements in global memory uncoalesced.

8.3 Method 2

The second approach aims to add more parallelism to the system and thereby improve Thread-Level Parallelism (TLP). This means instead of using just one thread to set up and solve each tri-diagonal system we will use one block containing as many threads as there are points is the grid.

8.3.1 Naive version

In the naive approach we simply start by letting each thread execute two device functions, `SetMA` and `SetMO`. This means that there is no longer need for a `for`-loop in the build kernels. Since the device functions in the solve kernels are a bit more complicated to parallelize they will in the naive implementation still be serial, meaning the first thread in each block performs all the calculations.

Some of the key values obtained by running the naive application through the NVIDIA Visual Profiler are shown in Table 8.14

Method	`xBuild`	`xSolve`	`yBuild`	`ySolve`
Duration (ms)	0.359	11.914	1.546	11.955
Requested gld throughput (GB/s)	73.16	1.5	13.81	1.51
Requested gst throughput (GB/s)	16.25	0.57	3.77	0.57
gld throughput (GB/s)	87.36	23.4	223.14	23.35
gst throughput (GB/s)	20.32	2.05	4.73	2.28
gld efficiency (%)	53.2	6.4	6.2	6.4
gst efficiency (%)	80	27.8	80	25
L1 glb hit rate (%)	71.7	25.1	16.4	25.2
IPC	0.581	0.157	0.118	0.157

Table 8.14: Metrics and events from NVIDIA Visual Profiler for the naive approach `S2N`.

It is clear that there is a huge difference in the performance of the kernels. In particular the solvers are performing horribly, see Figure 8.10. This is not surprising, since only one thread in each block is solving the system. Obviously, more parallelism have to be added to increase performance.

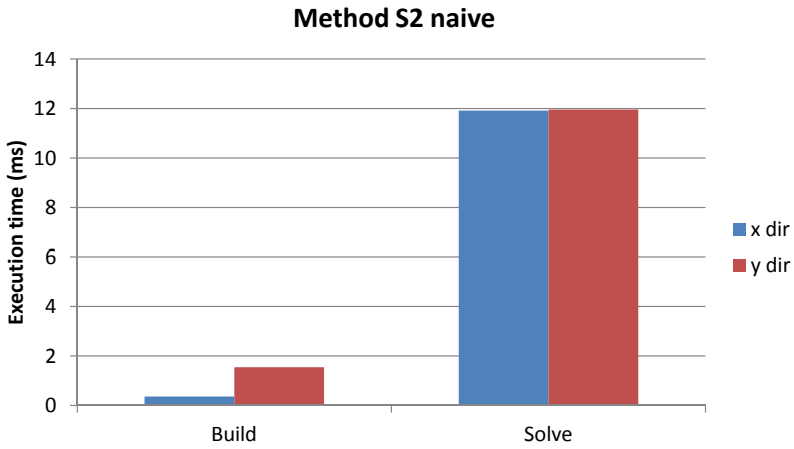


Figure 8.10: Execution time overview for method `S2N`.

In the build kernels, we see some similarity to Section 8.2.2 for `S1` method. The `xBuild` actually performs pretty good with high global load and store efficiency and high requested global load throughput on 87.92 GB/s. The `yBuild` on the other hand is not performing very well. It is 3.9x slower than `xBuild` even though they perform the same calculations. As it was the case for `S1N` this is due to uncoalesced global memory reads, which is clear given the very low requested global load throughput and the unrealistic high global load throughput[4]. This can also be realised when looking at the way the threads access the data, as shown in Figure 8.11

[4]Compared to the theoretical possible bandwidth the global load throughput is way to high. An explanation of this i given in Section 7.2.

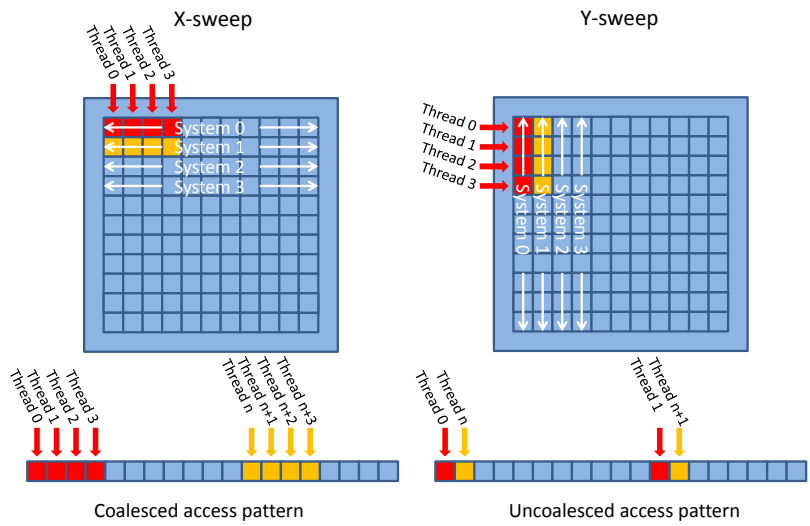


Figure 8.11: Access pattern for `S2N` for an x- and y-sweep.

Even though the kernels have a lot of issues they still outperform the C implementation by a 2.2x speedup over 100 time steps.

8.3.2 Version 2

The solver has to be added more parallelism to improve the overall simulation time. As discovered in Section 8.2.4, the calculation in the device function `LcElim` can be performed independently. Therefore, it is possible to go from a complexity of $O(n)$ for a single thread to $O(1)$ for n threads by adding more parallelism. Consequently, it is an obvious place to let more threads work simultaneously. On the contrary the Thomas algorithm, described in Section 5.2, is a serial algorithm and cannot be parallelized. Therefore, will this part of the kernel remain serial for now.

The uncoalesced access due to `yBuild` does not seem like an important bottleneck right now, since the solver is taking up most of the time. However, we foresee that it will be a bottleneck later on so it will be fixed as we did in Section 8.2.2. Thus the same sweep is used twice with a transpose in-between. The only difference is that now the x-sweep is loading coalesced and therefore we end up with the flowchart as illustrated in Figure 8.12

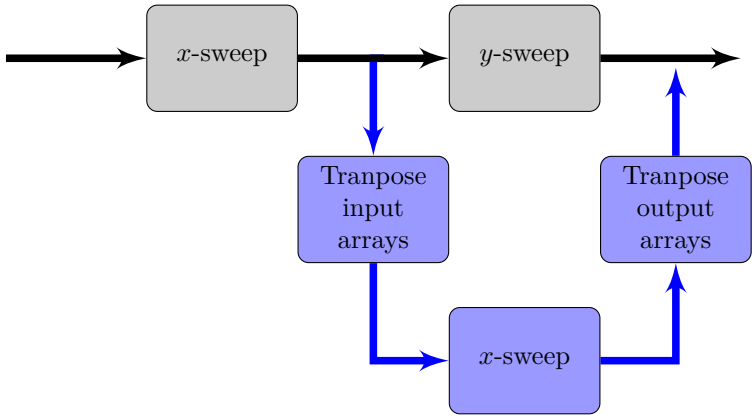


Figure 8.12: Flowchart through one simulation time step in the naive `S2N` and modified `S22` approach.

Implementing this we get from NVIDIA Visual Profiler the data shown in Table 8.15

Method	`xBuild`	`xSolve`
Duration (ms)	0.359	5.224
Requested gld throughput (GB/s)	73.16	3.44
Requested gst throughput (GB/s)	16.25	1.3
gld throughput (GB/s)	87.36	32.47
gst throughput (GB/s)	20.32	3.17
gld efficiency (%)	53.2	10.8
gst efficiency (%)	80	41.6
L1 glb hit rate (%)	71.7	33.7
IPC	0.581	0.155

Table 8.15: Metrics and events from NVIDIA Visual Profiler for the `S22` approach.

It should be mentioned that transposing all 7 vectors twice has a total runtime of 0.2 ms, but even through transposing nearly take the same amount of time as building the system (`xBuild`) it is definitely worth doing when considering the time saved on the `yBuild`. However, transposing the 7 vectors could later on become a performance limit. The execution times for the naive and optimized version `S22` are shown in Figure 8.13

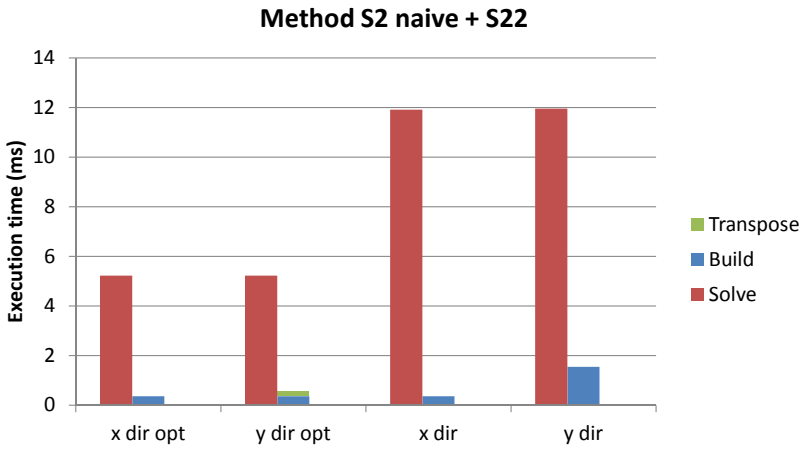


Figure 8.13: Execution time overview for `S2N` and `S22`.

We see that the optimized `xBuild` (`yBuild` and transpose) is 2.8x faster than the naive `xBuild`. By parallelizing the `LcElim` function the solver got a 2.3x speedup.

Performance evaluation

As shown in Table 8.16 the optimization results in a 5x speedup compared to the C implementation over 100 time steps.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	5.534 s			
S2N	2.552 s	2.2x	3.13 GB/s	1.08 Gflops
S22	1.117 s	5.0x	7.16 GB/s	2.48 Gflops

Table 8.16: Comparing execution time of the CPU against `S2N` and `S22` on a 256 × 256 grid over 100 time steps in double-precision.

When looking at both Table 8.15 and Figure 8.13 it is clear that the largest performance impact is the improvement of `xSolve`. However, `xSolve` still have the greatest performance issues with only 10.8% global load efficiency and 3.44 GB/s requested global load throughput. This is also why we only get an effective bandwidth of 7.16 GB/s which is only 4.37% of the theoretical value.

8.3.3 Version 3

To optimize further it is crucial to add more parallelism into solving each tri-diagonal system matrix. Unfortunately, this is not possible with the Thomas algorithm, since it is a serial algorithm as described in Section 5.2. Therefore, we have to use a different solution algorithm to increase performance. In [47] they found that the Parallel Cyclic Reduction (PCR) algorithm was the fastest of the standard algorithms tested. For that reason this algorithm will be implemented. The PCR algorithm is described in details in Section 5.3.

The special data structure, where mass and momentum equations are split in each vector, means that the first elimination is special compared to the rest of the eliminations. The first iteration has to use values from the mass equation vectors to perform elimination in the momentum equation vectors and vice versa. In all the other iterations are values from the same vector used to perform the eliminations. See Section 5.3 for further description of the work flow.

This makes our implementation of `PCR` special in two ways. First it has a special first iteration and secondly each thread will perform elimination in both the mass and the momentum equation vectors in each iteration. Consequently the work load is double for each thread, but only half the number of threads are needed to solve the system. However, this fits perfectly with the threads already available, since we have `sizeX` threads available to solve a $2 \cdot$`sizeX` $\times 2 \cdot$`sizeX` system and therefore only the algorithm has to be changed. When the `PCR` algorithm as implemented in Section 3.7 in the Source Code Booklet is used instead of the Thomas algorithm we get the results shown in Table 8.17 from the profiler.

Method	`xBuild`	`xSolve`
Duration (ms)	0.359	0.767
Requested gld throughput (GB/s)	73.16	117.63
Requested gst throughput (GB/s)	16.25	47.09
gld throughput (GB/s)	87.36	234.8
gst throughput (GB/s)	20.32	58.59
gld efficiency (%)	53.2	52.1
gst efficiency (%)	80	80.4
L1 glb hit rate (%)	71.7	48.5
IPC	0.581	0.739

Table 8.17: Memory metrics and events from NVIDIA Visual Profiler for the `S23` approach.

The `PCR` algorithm clearly improved the performance of the solver dramatically! It went from 5.224 ms to 0.767 ms, resulting in 6.8x speedup.

Performance evaluation

The application is now 22.7x faster than the C implementation as shown in Table 8.18.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	5.534 s			
S23	0.242 s	22.9x	20.16 GB/s	5.69 Gflops

Table 8.18: Comparing execution time of the CPU against `S24` on a 256 × 256 grid over 100 time steps in double-precision.

We also now achieve 20% utilization of the theoretical available bandwidth. Further we see that IPC has increased from 0.155 to 0.739 instructions per clock, which indicate that we do more instructions in less time. We see that even though the runtime has been improved a lot we still only get a 13% utilization of the theoretical bandwidth. So we expect that more performance can be gained.

8.3.4 Version 4

The solver is still taking up around $\frac{2}{3}$ of the execution time, so in order to achieve faster application runtime it is important to look at the solver. This will be done in Section 8.4. For now, however we will make sure that the the `xBuild` is performing satisfactorily.

In order to increase the performance of `xBuild`, we look at the following

- Reduce global memory stores by not storing `ama`, `bma` and `cma`.
- Perform `LcElim` immediately when building the system, so we do not have to read and write to `lmo` and `rmo`.
- Improve the calculations for `bmo`.
- Remove unnecessary `if`-statements.
- Perform the calculations so we improve ILP (latency hiding/register dependencies)

This is all steps that has been described throughout optimization of method `S1` in Section 8.2, so we will not describe it further here.

An optimization step we also perform is to investigate whether or not we could increase global load efficiency by loading the global values to shared memory, registers or just cache the values by using the L1 cache, similar to what is done in Section 8.2.3. The results here differs a bit from what we experienced then. When trying the different approaches we get the execution times shown in table 8.19 for 100 time steps.

Method	Before opt.	Shared mem.	Registers	48 kB L1	No L1
Duration	218.17 ms	222.78 ms	218.6 ms	207.95 ms	235.96 ms

Table 8.19: Different approaches to achieve better global load efficiency for `S24`.

We see that all the efforts with introducing shared memory and registers for trying to improve performance actually makes the kernel slower. Yet caching in L1 is important. Using 48 kB L1 instead of 16 kB decrease the execution time by 5% for the entire application and turning L1 cache off increases the execution time by 8.2%. Thus one simple function call return a 5% free performance boost. Consequently, we do not try to improve the global load efficiency by using shared memory or registers, but simply increase the L1 cache size to maximize performance.

In general we notice that all our performance optimizations have less impact than in `S1`. Especially latency hiding gives almost no performance boost. However, this is not totally surprising, since we for this kernel achieve an occupancy of 31%. Thereby the GPU have more freedom to perform latency hiding it self.

With all this optimization steps implemented, we get the results shown in Table 8.20 and 8.21 from the profiler.

Method	`xBuild`	`xSolve`
Duration (ms)	0.231	0.693
Requested gld throughput (GB/s)	69.55	111.12
Requested gst throughput (GB/s)	10.53	47.86
gld throughput (GB/s)	135.66	219.12
gst throughput (GB/s)	13.13	59.78
gld efficiency (%)	50.9	50.6
gst efficiency (%)	80.2	80.1
L1 glb hit rate (%)	84.4	68.4
IPC	0.635	0.831

Table 8.20: Memory metrics and events from NVIDIA Visual Profiler for the `S24` approach.

We see a quite significant performance increase, especially in `xBuild` where we get a 1.6x speedup. Looking at the metrics/events received from the profiler in Table 8.20 we see that only L1 global hit rate and instructions per cycle have improved. The other values decrease and that is simply because we read/write less and perform less instructions.

For instance, we see in Table 8.21 that 27% less instructions is issued for `xBuild`. We also see that replayed instructions and the ratio of divergent branches both drop for `xBuild24` due to the optimized `if`-statements.

Method	xBuildS23	xSolveS23	xBuildS24	xSolveS24
Inst. issued	1,943,856	6,092,051	1,416,018	5,417,627
Inst. executed	1,697,758	5,240,255	1,281,817	4,815,039
Threads inst. exec	54,034,986	161,692,925	40,814,580	148,143,869
Branch	101,238	276,224	63,630	261,888
Div branch	793	2085	256	2341
Ratio div branch	0.8%	0.7%	0.4%	0.9%
Control flow div	0.5%	3.6%	0.5%	3.9%
Replayed inst	12.7%	14.8%	9.5%	11.2%

Table 8.21: Instructions metrics and events from NVIDIA Visual Profiler for the S23 and S24 approach.

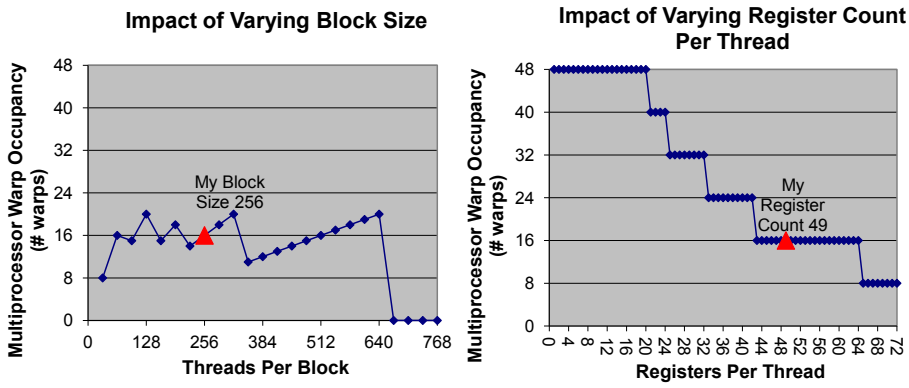
Performance evaluation

We have now reached more than 33x speedup compared to the C implementation as shown in Table 8.22. The utilization of the bandwidth still seems a bit low with only 30% of the theoretical value so it is possible that even higher performance can be obtained.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	5.534 s	-	-	-
S23	0.164 s	33.62x	29.66 GB/s	8.37 Gflops

Table 8.22: Comparing execution time of the CPU against S24 on a 256 $\times$ 256 grid over 100 time steps in double-precision.

It should be said that xBuildS24 uses 49 registers per thread. Since there are only 32 K 32-bit registers available per multiprocessor this means that only a system of size 668×668 can be run unless the compiler option $-$maxrregcount is used to control the number of registers used per thread. This is also shown in Figure 8.14a.



(a) (b)

Figure 8.14: Impact on occupancy when varying block size 8.14a and registers per thread 8.14b for `S24`.

The compile option might however cost some performance which will be investigated in Section 9.2.

The many used threads also means that we can only achieve an occupancy of 33%. Even though this is a lot higher than for `S1` it still could have an impact of how well the GPU can latency hide. In Figure 8.14b it is shown that we could increase the occupancy to 50% if we could get the registers per tread down to 42. We where however not able to decrease the number of used registers without losing performance.

`xBuild` has now been satisfactorily optimized. In order to achieve higher performance it is therefore necessary to focus on the solver, as previously mentioned. This will be done in Section 8.4.

8.4 Solver

In Section 8.3.4 we saw that the performance of the solver was crucial for the overall run time. In fact, even with a parallel solver it is responsible for 65% - 75% of the execution time! For this reason, we will investigate how the implemented PCR solver can be improved and implement a hybrid CR-PCR solver. See Chapter 5 for more detailed overview of the different algorithms and their workflow. The source code is attached in Section 3.7 and 3.8 for PCR and CR-PCR, respectively, in the Source Code Booklet

8.4.1 Parallel cyclic reduction

The PCR solver which was naively implemented in Section 8.3.3 has several differences compared to the Thomas algorithm solver. The thing that immediately springs to mind is of course that it is a parallel algorithm, thus more threads can work simultaneously. The disadvantage is that the elements in the vectors are updated a lot more times and therefore a lot more work is performed. In fact, for n unknown, the Thomas algorithm require $O(n)$ operations and steps compared to PCR that require $O(n \log_2(n))$ operations but $O(\log_2(n))$ steps to finish, if n processors are available.

We have seen that the GPU (Fermi architecture) is very efficient at optimizing global loads and stores by caching in L1 cache. Nevertheless, in this case, it might be beneficial to load the vectors into shared memory to do all the updates there and hereafter return the result back to global memory. Although simply increasing the L1 cache size might also have a positive impact on the performance. We try both approaches and the results from the NVIDIA Visual Profiler can be seen in Table 8.23.

Method	PCR naive	PCR w. 48 kB L1	PCR w. shared mem.
Duration (ms)	0.767	0.614	0.506
Requested gld throughput (GB/s)	117.63	125.56	4.82
Requested gst throughput (GB/s)	47.09	54.08	1.92
gld throughput (GB/s)	234.8	253.58	2.34
gst throughput (GB/s)	58.59	67.55	9.64
gld efficiency (%)	52.1	50.4	50
gst efficiency (%)	80.4	80.1	82
L1 glb hit rate (%)	48.5	67.6	45.6
Shared memory bank conflicts	-	-	0
Shared memory load	-	-	328,704
Shared memory store	-	-	145,408
IPC	0.739	0.839	1.044

Table 8.23: Memory metrics and events from NVIDIA Visual Profiler for the different PCR solver optimizations.

It is clear that both optimization steps has a big impact on the performance of the solver. Increasing the L1 cache to 48 kB gives 1.25x speedup and using 48 kB shared memory gives 1.52x speedup compared to the naive implementation. Further we see that the instructions per clock is rather good. As expected, the global memory throughputs are a lot smaller for the shared memory implementation, since a lot less data are transferred to and from global memory. It should be mentioned that the low global load efficiencies does not correspond to uncoalesced global memory access, but is simply due to the system size as described in Section 7.3.1.

We now see that shared memory really has a positive impact compared to just increased L1 cache size. However, using shared memory does have its downsides. For instance, we can only allocate 48 kB of shared memory, which corresponds to 6,144 elements of data type `double`. Because we need to store 8 vectors in shared memory the maximum possible system size contain only 768 elements. This is a 25% smaller system than the method normally would be able to solve.

To improve this we would have to not use some of the 8 vectors, which is not possible with this algorithm, since elements in all vectors are updated throughout the algorithm. This however motivates to use the CR-PCR hybrid algorithm presented by [47].

8.4.2 Cyclic reduction + parallel cyclic reduction hybrid

We will now investigate how the Cyclic Reduction - Parallel Cyclic Reduction hybrid algorithm can be beneficial to use on this system. The algorithm and its workflow is described in detail in Section 5.4.

As previously described the CR-PCR algorithm only uses every other element in the system for the first and the last iteration. This implies that when using this algorithm, there is no reason to use vectors to store `ama`, `bma` and `cma`, since they are defined as constant and never changes though the algorithm.

On top of that, for each of the CR steps of the algorithm, we halve the work that has to be done. Performing just a single CR step will halve the work of all future steps only at the cost of a single iteration more at the end. Hence each thread will only eliminate a single element in each iteration instead of two. The problem with performing CR steps is that for each iteration the bank conflicts is increased by a factor 2 due to the larger stride. In [13], they find that a pure CR algorithm can almost perform as well as the CR-PCR hybrid approach by [47]. This is done by allocating a new shared array of smaller size in each iteration to store the intermediate systems in order to avoid bank conflicts. However, since our vectors are divided into mass and momentum equations, we have already allocated an array for the intermediate system for the first iteration. For this reason we can perform a single CR step without having to worry about bank conflicts.

All together it looks very attractive to implement the CR-PCR Hybrid. This is due to the smaller amount of vectors that has to be used, the less work per thread, and (at least for the first step) no bank conflicts in the CR steps. We will implement this version and test how it performs when using 16 kB L1 cache, 48 kB L1 cache and with shared memory as we did for the PCR algorithm.

The implemented solver takes a single CR step before switching to PCR. The results from the three different versions can be seen in Table 8.24.

Method	PCR Naive	PCR w. 48 kB L1	PCR w. shared mem.
Duration (ms)	0.430	0.404	0.275
Requested gld throughput (GB/s)	123.4	131.22	7.93
Requested gst throughput (GB/s)	45.29	48.16	3.17
gld throughput (GB/s)	238.28	247.11	15.85
gst throughput (GB/s)	56.41	59.99	3.86
gld efficiency (%)	52.8	52.4	50.1
gst efficiency (%)	80.3	80.3	82
L1 glb hit rate (%)	66	78.9	32.1
Shared memory bank conflicts	-	-	0
Shared memory load	-	-	223,232
Shared memory store	-	-	88,064
IPC	0.74	0.784	0.921

Table 8.24: Memory metrics and events from NVIDIA Visual Profiler for the different CR-PCR solver optimizations.

It is immediately clear that the CR-PCR hybrid algorithm significantly outperforms the PCR algorithm. In fact, the naive CR-PCR uses 15% less time than the fastest PCR implementation. Again we see that using shared memory is the fastest approach with a 1.84x speedup compared to the fastest PCR implementation. This is because the algorithm simply performs less work compared to the PCR approach, which also is clear from the fact that 32% less shared memory loads and 39% less shared memory stores are performed.

8.4.3 Performance evaluation

The speedup for the entire application when using the fastest PCR and the fastest CR-PCR implementations are shown in Table 8.25.

Method	Duration	Speedup	Bandwidth	Gflops
CPU	5.534 s			
`S24` PCR	0.168 s	32.9x	47.62 GB/s	16.46 Gflops
`S24` CR-PCR	0.130 s	42.6x	61.54 GB/s	21.27 Gflops

Table 8.25: Comparing execution time of the CPU against `S24PCR` with shared memory and `S24CR-PCR` with shared memory on a 256×256 grid over 100 time steps.

Clearly, this is a significant speedup for such a small system even though the effective bandwidth when using the CR-PCR solver is only 37.6% of the theoretical value.

The reason why we for the 2nd method (`S2`) is capable of achieving such

high speedups for small system sizes compared to the 1st method (`S1`), is simply because we have added more parallelism and thereby divided the work into much more threads. Hence we can better utilize the TLP and thereby increase the performance. The execution time for all the different parallel solvers is shown in Figure 8.15.

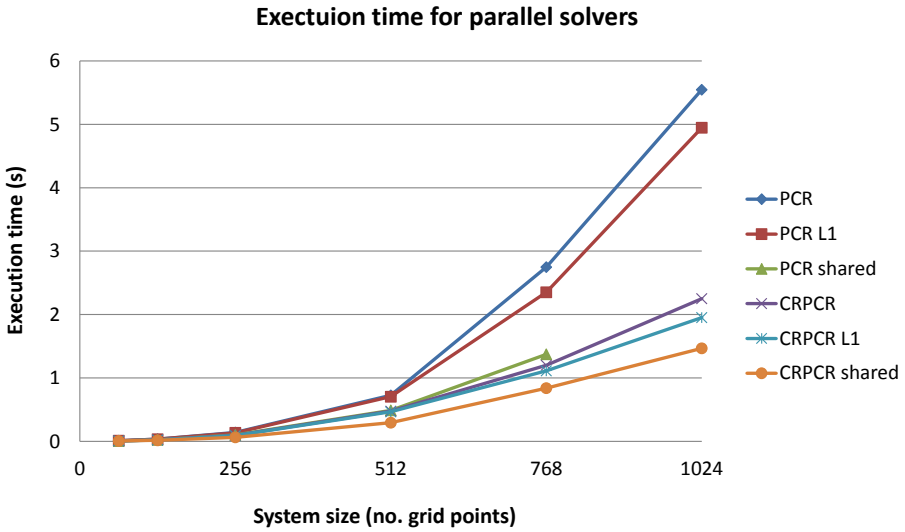


Figure 8.15: Effective bandwidth for all the different parallel solvers.

It is clear that the CR-PCR hybrids outperform the PCR and that using shared memory is very beneficial. It should also be noticed that the CR-PCR implementation is not limited by the shared memory size like the PCR approach. This is due to the smaller amount of vectors we allocate, because we now only have to allocate 5 vectors (`amo`, `bmo`, `cmo`, `dma` and `dmo`). Thus each vector can contain $48\text{ kB}/(8 \cdot 5) = 1{,}228$ double elements per vector. So we are back to the limit of a 1024×1024 system size, since the maximum amount of threads per block is 1024. In figure 8.16 is the effective bandwidth for the different solvers shown.

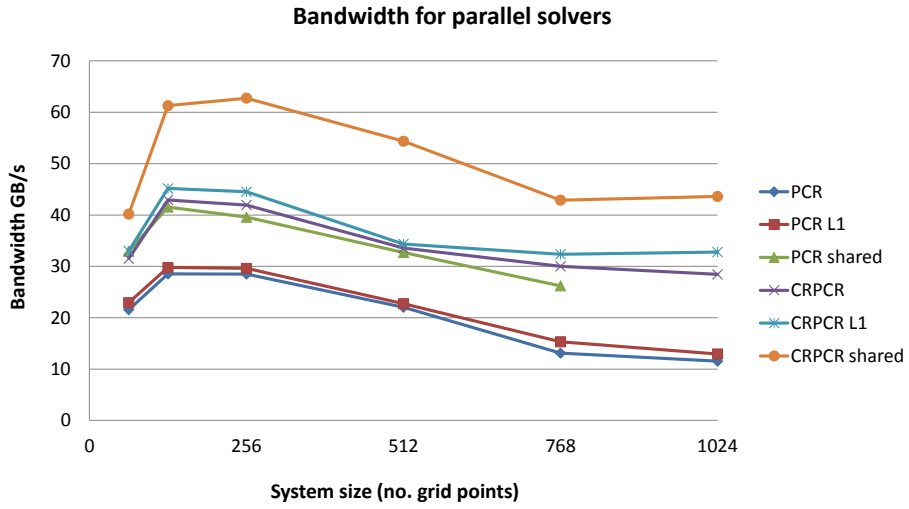


Figure 8.16: Effective bandwidth for all the different parallel solvers.

We see that the solvers peak in performance already on a 256×256 system. This is not that surprising because of the algorithms used. Since the PCR algorithm (and CR-PCR because we only take a singe CR step) requires $O(n\log(n))$ work to be done and since it has to be performed n times for a sweep we end up with a $O\left(n^2\log(n)\right)$ work per sweep. However, since we can solve the systems in parallel the runtime is only $O(\log(n))$ given we have n^2 processors available. This means that the bigger the system gets the more will our algorithm suffer from lack of processors. This however could be made better for the hybrid algorithm if it took some additional CR steps reducing the system to a size where we would have enough processors to efficiently run the PCR algorithm. This will, however, start causing a 2^k way bank conflicts, where k is the number of additional CR steps, if the updating for the elements are done in place. This implies that even though we halve the work to do in each step, we double the bank conflicts and no additional performance would be gained. Therefore, to achieve the better performance it would be necessary to find a way to remove those bank conflicts.

In [47], they simply choose to disregard the occurring bank conflicts in the CR steps. Then to avoid bank conflict in the PCR steps they load the intermediate system into a new array. In [13], they use a new array to store the results from each CR steps and thereby avoid bank conflicts but at the cost of more shared memory. These are both approaches that could be used to make additional CR steps beneficial, but this will not be investigated further in this project.

Chapter 9

Performance Results

In this chapter will the performance of the developed kernels be investigated in further detail. This involves investigating the performance as a function of the problem size, the optimal size of thread blocks for `S1`, the performance of the two parallel methods and finally the impact in performances, when switching from double- to single-precision on the two most optimized kernels. All tests are performed with the test configurations described in 7.3.1 and in double-precision when nothing else is stated.

9.1 Method 1

9.1.1 Block sizes

For `S1` we have the option of varying the block size. Increasing the block size will (if the system is large enough) allow for more resident threads on the multiprocessor, achieving better occupancy. However, when increasing the number of threads per block one needs to solve a larger system to maintain enough blocks to distribute over all multiprocessors. It is therefore not obvious what the optimal block size is for a given problem size, but the size should always be a multiple of 32 to achieve maximum warp utilization. To investigate the performance differences for different block sizes is the optimized kernel, `S14`, executed for different block and system sizes, see Figure 9.1

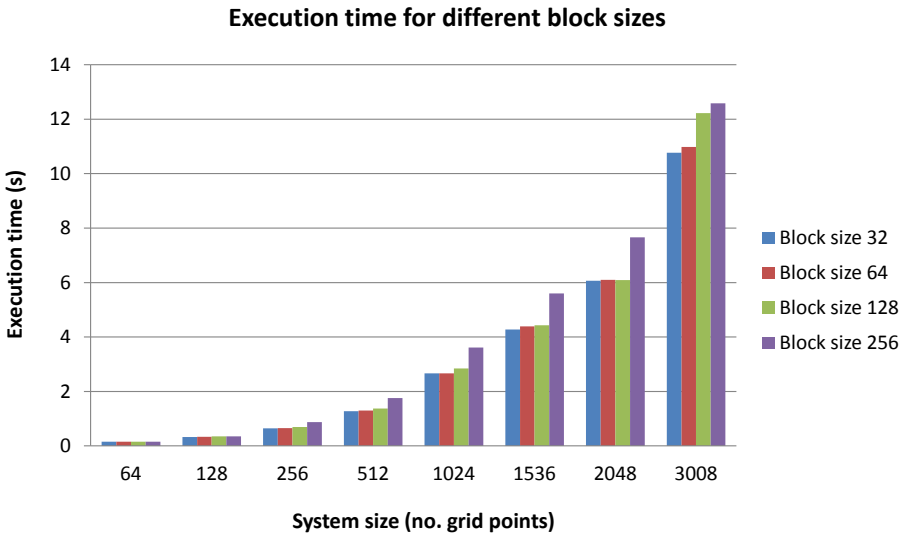


Figure 9.1: The execution time for different block and system sizes.

Clearly varying the block size has very little impact on the performance. However, we see that smaller block sizes performs slightly better compared to the larger blocks for all the system sizes. This is because we never hit the limit of 8 resident blocks per multiprocessor even when our block size is only 32. Thus increasing the block size does not imply better occupancy before we have a larger system. Furthermore, smaller block sizes will give the GPU more freedom to distribute the blocks over the streaming processor array and thereby even out the workload on each multiprocessor to insure that the GPU is as busy as possible.

The block size of 32 and 64 threads performs almost identically. This means auto tuning the device for optimal performance based on block size becomes very easy. A block size of 64 can simply be used for all system sizes and achieve high performance, since it is assumed to be the best for lager systems. For this reason will all results from now be achieved using 64 threads per block.

9.1.2 Performance test

For `S1` is the problem size very important for the performance of the application. Even if a block size of 32 is used, we still need at least a 512×512 system to just have a single block per multiprocessor. For this reason, the application clearly will perform better for larger systems. However, we are limited on the problem size simply by the limited available DRAM memory on the hardware. In Section 8.2.5.1 (8.2.6), we found that the optimized application, `S1`, could calculate a 21% larger system without using more memory compared to the naive application. For that reason will `S14` be tested on larger systems compared to the other; to verify that `S14` can simulate larger systems and to illustrate that the performance still increases for larger systems,

Figure 9.2 illustrates the execution time for all the `S1` versions for three chosen systems sizes and the C implementation.

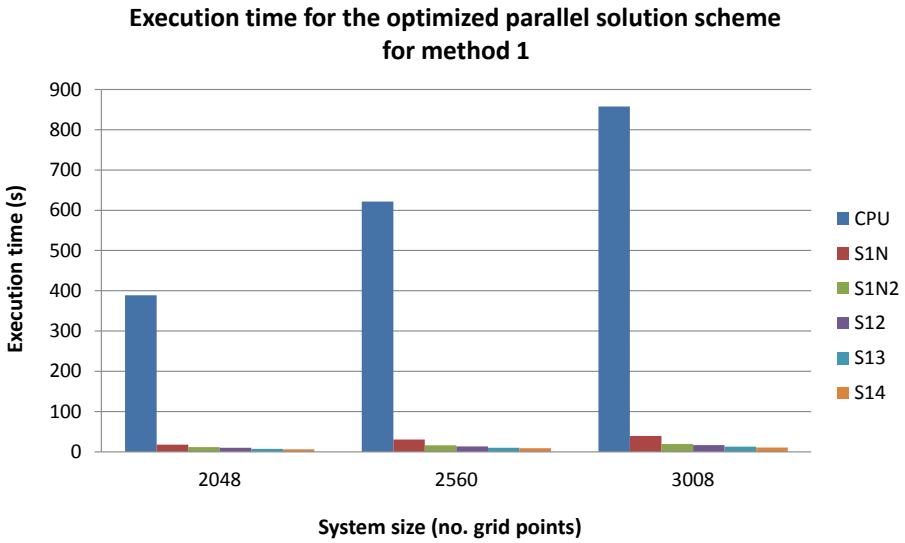


Figure 9.2: Execution time for three chosen systems sizes 2048×2048, 2560×2560 and 3008×3008 using approach `S1` and the C implementation.

We see that the CPU execution time is extremely high compared to all the different parallel implementations. In fact, the difference is so large that we almost cannot see the executions time of the parallel implementations. For better illustration are the speedups for the different optimization steps as function of the problem size shown in Figure 9.3.

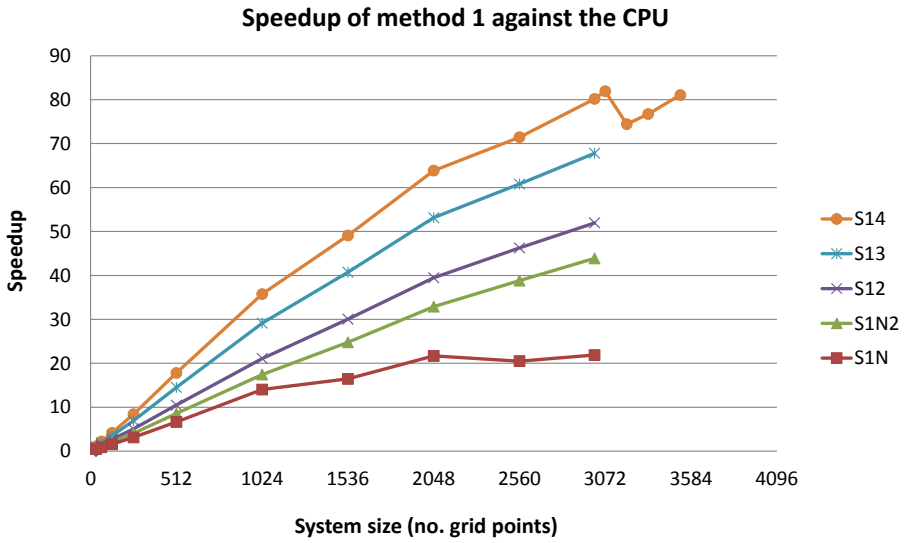


Figure 9.3: Speedup of all `S2` implementations.

As expected, we see that the speedup increases as the problem size increases and that each performed optimization step have resulted in a performance increase. We achieve a speedup of more than 80x compared to the C implementation for problem size 3008×3008 for `S14`. Thus we can run a simulation in one hour, where before it was necessary to wait 80 hour to obtain the same results. Or put in another way, we can solve a 3008×3008 system twice as fast as the CPU can solve a 512×512 system. For `S14` we also see that one can simulate larger systems compared to naive versions.

In Figure 9.4 is the achieved effective bandwidth illustrated as a function of the system size.

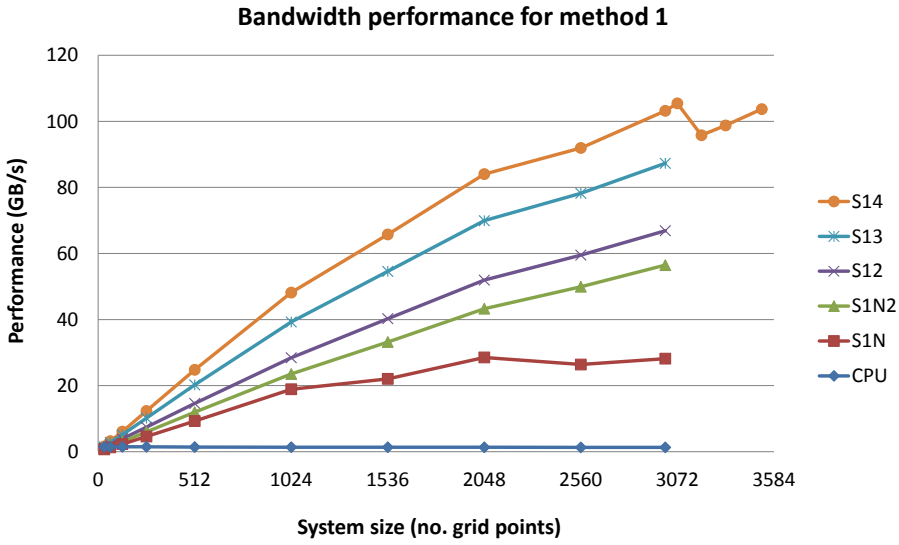


Figure 9.4: Effective bandwidth of all `S2` implementations.

We obtain better bandwidth utilization as the system size is getting larger. E.g., for a problem size of 3008×3008 we achieve 103 GB/s, which is about 63% utilization of the theoretic bandwidth, while the `bandwidthTest` obtain 80% utilization. Thus we utilize the available bandwidth on the GPU fairly good and it is obviously the available bandwidth which limits the performance. The drop in performance for `S14` around system size 3072×3072 is due to the non-uniform work distribution, as described later.

For the CPU, one can see a plain effective bandwidth. In fact, it drops for larger systems due to the memory hierarchy on the host.

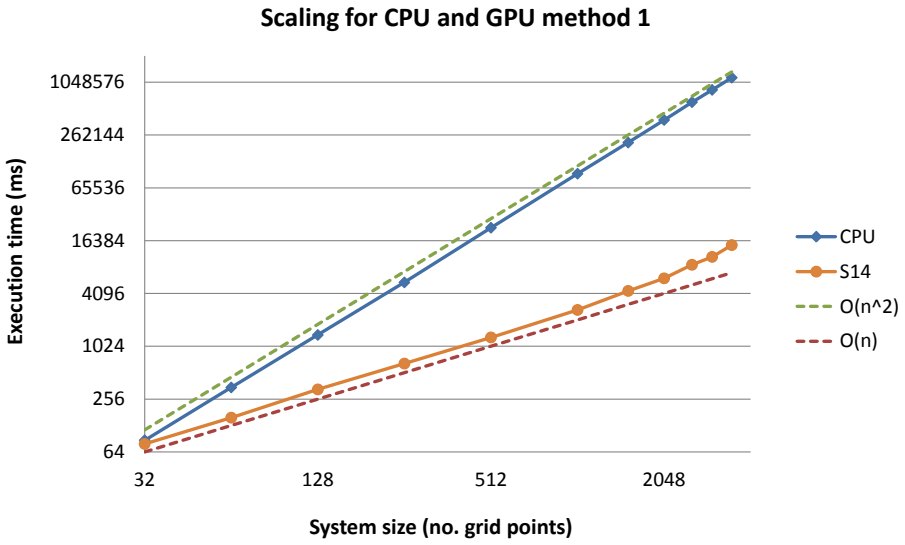


Figure 9.5: Scaling of the execution time for the CPU and `S14` in log-log plot.

Figure 9.5 illustrates the scaling of the execution time for the CPU and `S14` in a log-log plot together with $O(n)$ and $O(n^2)$. Recall that the gradient of a polynomium is the slope in a log-log plot. As expected, we see that the CPU scales quadratic and `S14` scales linearly. This is expected, since the sequential C implementation run through all grid points one by one by using a nested `for` -loop. On the other hand the parallel solution scheme, `S14`, run through all the grid points in linear time given n processor, where n is either the number of rows or column in the system. This is analogue to what is seen in both Figure 9.3 and 9.4 for `S14`. For systems size 32×32 to 1024×1024, are not all multiprocessors doing work, because there are not enough blocks/threads. Therefore, when increasing the system size in this interval, we actually just distribute more work to processors that before did not work. This verifies that it should scale linearly, since only the increased workload per block will influence the execution time.

For system size 1024×1024 and up to 2048×2048 we see the slope decreases a bit and again from 2048×2048 and so forth. However, we still obtain linear scaling, although we cannot add the more work to new processors. The reason why this is possible is due to one of the many advantages with parallel programming on the GPU, namely latency hiding. It is simply possible for the GPU (the multiprocessors), with more work, to hide the latency and thereby maintain the linear scaling.

It is obviously not possible for the GPU to keep this forever. There is a limit to the amount of work, which can be performed at no extra cost. This

is also why we see the slope decreasing more and more for both the effective bandwidth and speedup and thereby it starts scaling "less linearly" as shown in the end of Figure 9.5. When `S14` simulates the largest systems one can also see that it is difficult for the GPU to hide the extra work. We get a performance drop at 3200×3200. The difference between that system size and 3072×3072 is the non-uniform work distribution over the streaming processor array. For a 3072 system all multiprocessors have 3 blocks each, but for a 3200 system there are 2 more blocks. Thereby there are two multiprocessors which have 33% more work than the others. Thus at some point in time the available resources on the GPU will not be fully utilized. Consequently, the drop in performance will happen when the work is not evenly distributed over the streaming processor array and the impact will be more significant for larger system sizes. However, the performance still increases, which indicates that we have not peaked yet, but after all will the performance increasing be less compared to the smaller system sizes.

The performance limit for this application was found in Chapter 2 and Section 7.2.1. Here we saw that the application was memory bounded, rather than compute bounded or bounded by a low parallel fraction. Thus we cannot continue with linear scaling, since the threads cannot calculate faster than they transfer data. Hence it behave as expected.

9.2 Method 2

For `S2` the block size is always equal to the system size and therefore it cannot be varied for a given system. This method has also a limit on the problem sizes which can be solved, yet it is not due to the memory size as it was for `S1`, but the fact that the maximum block size on the Fermi architecture is 1,024 threads per block.

9.2.1 Performance test

Running the different `S2` versions we achieve the speedups shown on Figure 9.6

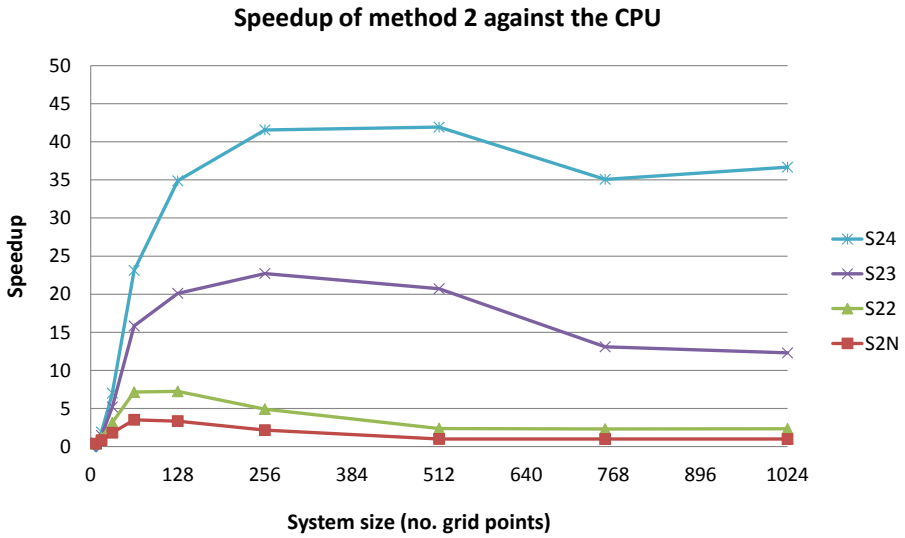


Figure 9.6: Speedup of all `S2` implementations.

We see that the naive version and `S22` have almost no speedup, especially when the problem size just gets a little larger. This shows how important it is to have a parallel implementation of the `LcElim` and the solver. The two other versions achieve quite high performance and `S24crpcr` achieve a peak of 42x speedup and an effective bandwidth of 66 GB/s, which is 41% of the theoretical bandwidth.

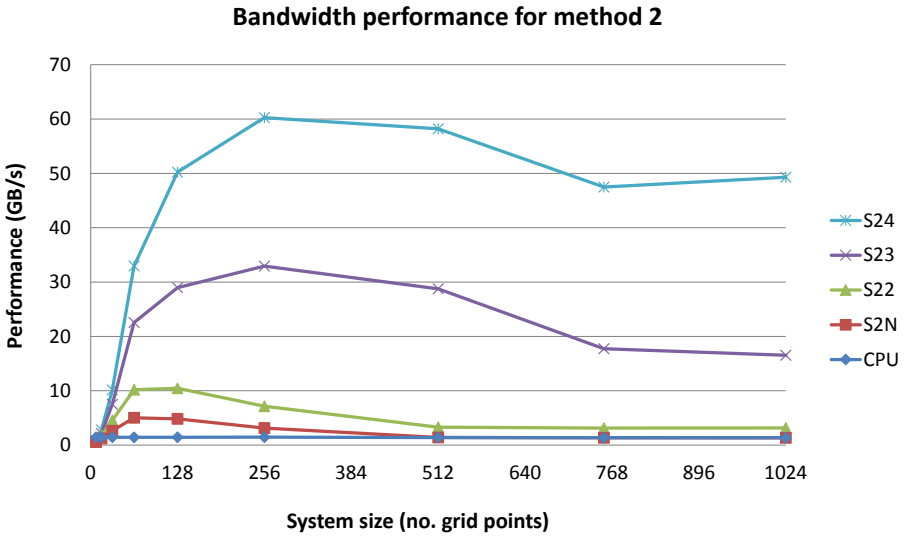


Figure 9.7: Effective bandwidth of all `S2` implementations.

We see a performance drop when testing for systems larger than 512×512. As described in Section 8.3.4, we have to use the compiler option `-maxrregcount` when solving systems larger than 668×668, because of the large amount of registers used by `xBuild`. This however turns out to limit the performance due to register spilling into local memory. When profiling `xBuild24` for a 756×756 system, we get the counters shown in Table 9.1

Method	`xBuild24`
L1 local load hit	378,268
L1 local load miss	297,276
L1 local store hit	29,032
L1 local store miss	703,664
instruction issued	15,119,170
L2 query read	3,903,097
L2 query write	3,049,300
gld request	598,928
gst request	90,720

Table 9.1: Profile counters for `xBuild` in `S24`.

From these values we can calculate the impact on the memory throughput by comparing L1 local load miss count to global load and store memory count: (gld request + gst request)/(2 · L1 local load miss). This implies a lot of traffic to global memory is due to spills, since the ratio of global memory to local

memory bus traffic is approximately 1.16 : 1. Thus almost half the traffic to global memory is due to register spills. On top of that, half of the spilling can not be contained in the L1 cache due to the 56% local L1 hit rate. Unfortunately there are not much to do about this, since we have not been able to reduce the number of registers used by each thread, non-caching by turning off the L1 cache does not help and we have already increased the L1 cache to 48 kB. The performance of `xBuild24` is shown in Figure 9.8

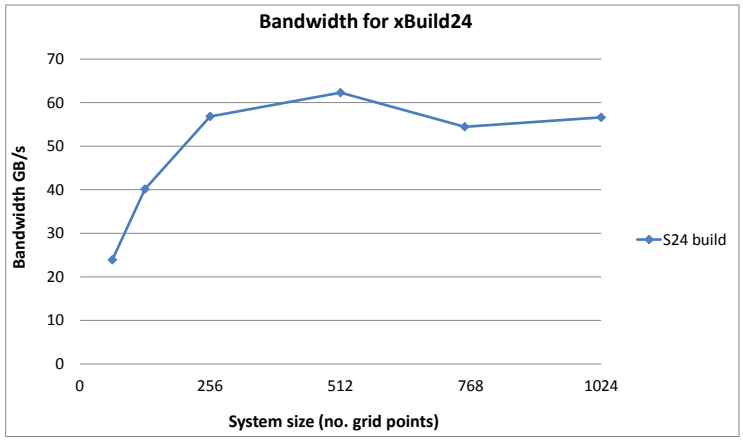


Figure 9.8: Effective bandwidth for `xBuild24`.

But even with the register spill, we still get a decent performance. Further with the new *Kepler* architecture from NVIDIA are there a lot more registers available per thread so this will no longer be an issue.

Another reason for the performance decrease for systems larger than 512×512 is the complexity of the solver as described in Section 8.4.3. Thus we see that the little performance drop for building the system due to the register spills is not the main issue of the lower performance for the application.

9.3 Merged Methods

It is clear that the performance of `S1` and `S2` peak in very different places. `S1` performs well for large systems and `S2` for small systems. This motivates to compare how the performance curve would look if we merged the two methods and thereby made it possible to simply switch between these two methods when it is beneficial. The speedup compared to the C implementation for the fastest `S1` and `S2` application is shown in Figure 9.9.

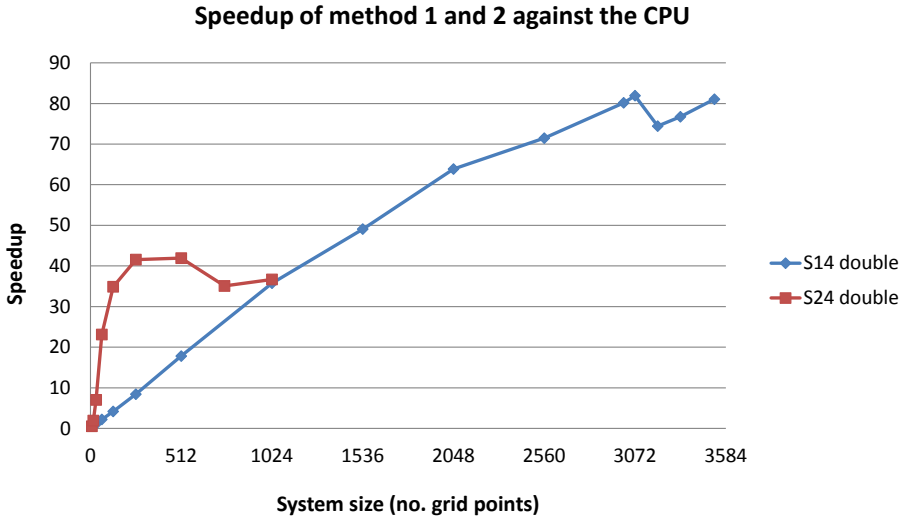


Figure 9.9: Speedup for `S14` and `S24` against the CPU.

The figure shows how well the two methods compliment each other. Just when `S2` reaches the system size limit, `S1` almost continuously takes over. This means that all system sizes between 128 and 3584 can be computed with these two methods with a speedup of minimum 35x and a maximum of 82x. This illustrates that developing these two different parallel approaches have not been for nothing, since one can apply the most suitable approach depending on the system size.

The quality that we can obtain good performance on all system sizes, is a very nice feature, that makes the application more robust. Notice that all these speedups are obtained with an error less than 10^{-12} compared to the C implementation and thereby compared to MIKE 21 HD.

9.4 Brief study of performance with single-precision

As motivated in Section 8.2.5, we will briefly investigate how the applications perform when all calculations are done in single- instead of double-precision floating point operations. Thus investigating the impact on GPGPU programming with these two data types.

To enable the opportunity of switching between single- and double-precision for either the host or device code in compile time, we have specified a macro identifier by using `#define`. On a modern CPU, calculation in single- or double-precision will almost take the same processing time. On a GPU are there still a big performance difference. Furthermore, `float` takes up only half the space compared to `doubles`, which implies that `S1` can compute even larger systems. In fact, we can compute a system of size 5179×5179, as described in Section 8.2.5.1. In Figure 9.10 the speedup for both `float` and `double` of the two methods are illustrated.

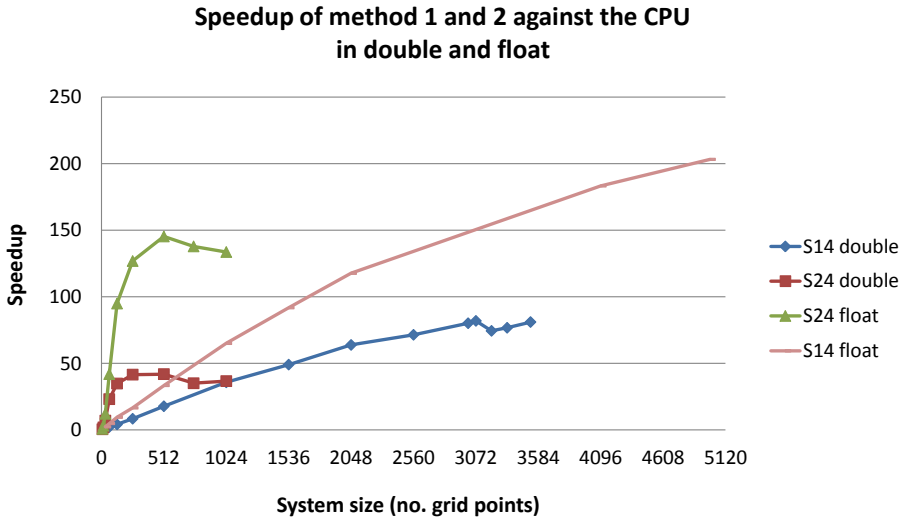


Figure 9.10: Speedup of `S14` and `S24` in single- and double-precision against the CPU.

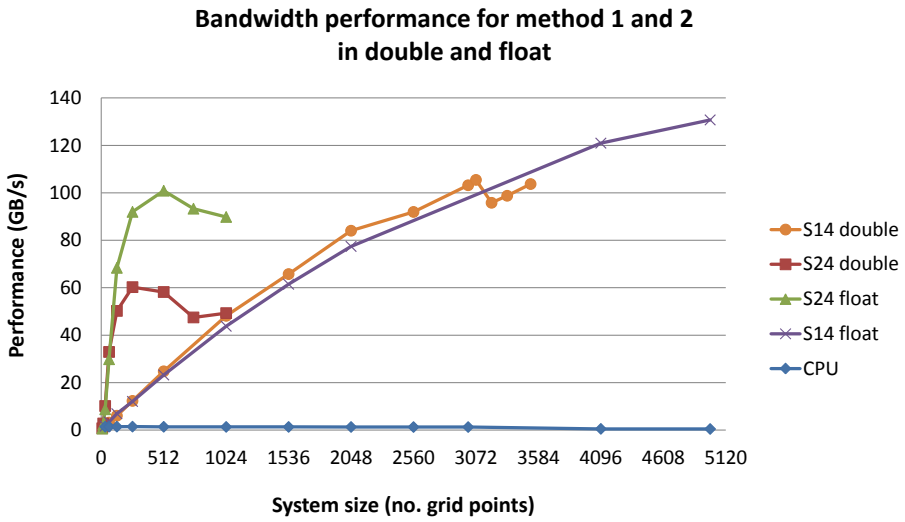


Figure 9.11: Effective bandwidth of `S14` and `S24` in single- and double-precision.

When using the data type `float`, it is important to switch the `pow()` and `sqrt()` functions in `SetMO` to single-precision functions instead, otherwise a lot of the performance would be lost. In fact, calling the heavyweight function `pow()` is more expensive than calling two `sqrt()` functions! Further are the application compiled with the following compiler options and flags

`-use_fast_math`	fast, but less precise math functions.
`-ftz=true`	flush denormalize numbers to zero.
`-prec-div=false`	fast, but less precise division.
`-prec-sqrt=false`	fast, but less precise square root.

We see that switching to `float` really has an impact on the performance! We now reach a maximum speedup for `S1` of 203x and for `S2` of 145x, and `S14` can solve much larger system sizes. `S14` can now solve problems that before took 7 days with MIKE 21 HD in less than 50 min! We also utilize the available bandwidth a lot better with a peak for `S1` of 130 GB/s and for `S2` 101 GB/s, which is 81% and 62% of the theoretical bandwidth on the NVIDIA GeForce GTX 590.

We see that the impact when solving in single- and double-precision is huge. For `S14` we see that for a 3072×3072 system size it goes from a speedup of 82x to 150x, i.e., nearly twice as fast. The single-precision version of `S24` reach 145x speedup compared to 42x for a 512×512 system, i.e., 3.5x faster. We do

not obtain 6x faster code due to the single- and double-precision floating point performance (see Section 7.3), as mentioned, because the application are limited by the bandwidth. The reason the speedup is higher than 2x is that the kernel benefit from more than just the reduced size of global transfers though lower register pressure, smaller amount of shared memory needed etc..

For `S1` it still seems to be able to get even higher speedups if more memory were available. Even though `S2` now uses a lot less registers and therefore no longer spills into local memory, we still see a drop at the end. This is due to the solver as described in Section 8.4.3. Now there is a performance gab between the two approaches. Thus if a system is slightly larger than 1024×1024 will only a speedup of 65x be achieved compared to 133x speedup, when a system smaller than 1024×1024 is executed. This is unfortunate, but we will not further try to improve this in this project.

Even though this really is an amazing performance boost it comes with a price. Obviously, when switching to `float` the precision drops simply because `float` is less accurate. In our tests the maximal error that has been recorded was on the 10^{-5} position. However, the error must be assumed to accumulate for more time steps, which also is what we experience. On the other hand, the error seems to be independent of the system size.

If DHI are interested in achieving this higher performance at the cost of some precision, a more thorough study of the error must be accomplished. Further, applying mixed-precision iterative refinement, as described in [13], could be performed, to achieve better performance and more accuracy on the GPU. Than executing the application implemented entirely in double- or single-precision, respectively. This implies that it is up to the user if it would be reasonable to sacrifice some precision to achieve better performance.

Chapter 10

Conclusion

In this project it has been shown that an efficient parallel solution scheme can be formulated for MIKE 21 HD and that the simulation speed can be drastically improved by using a GPU to accelerate the application. Two different approaches have been used to parallelize the solution scheme of MIKE 21 HD. The 1st method utilize that each tri-diagonal system can be build and solved independently, while the 2nd method also build and solve each tri-diagonal system in parallel.

Both approaches showed great performance increase compared to the sequential C implementation of MIKE 21 HD with a maximum of 42x and 80x speedup in double-precision and 145x and 203x speedup in single-precision for the two approaches respectively. For comparison can a 3072×3072 system be solved in double-precision on the GPU twice as fast as a 512×512 system on the CPU.

As a result of the two different approaches it is possible to obtain high performance even for small system sizes. However, for small systems it was necessary to apply a different solver algorithm than the Thomas algorithm in order to add enough parallelism and thereby utilize the hardware. We found that the hybrid approach of taking a single cyclic reduction step before switching to parallel cyclic reduction resulted in the highest performance of the investigated algorithms.

It is shown that the two developed approaches complement each other nicely, since the 2nd method is beneficial for small systems, while the 1st method is beneficial for larger system. Hence it is possible to apply the most advantageous method depending on problem size and thereby obtain a robust auto tuning of the application.

All results in the project can be obtained with a GPU which costs around 500$ and therefore no expensive hardware is necessary.

All results are validated against a sequential representative C implementation of MIKE 21 HD. Developing the C application turned out to be a very extensive task due to the size and complexity of MIKE 21 HD. For this reason has a larger part of the project, than estimated, been spend on understanding MIKE 21 HD and implementing the C application.

Conclusively, DHI will without a doubt be able to achieve a significant improvement in simulation speed and solve much larger systems in a reasonable amount of time, by using the applications developed in the project. The results are in fact far greater than expected by DHI at the beginning of the project. Hence we have developed an efficient GPU-accelerated MIKE 21 HD solver for shallow water fluid flow equations, which can be incorporated into the existing MIKE 21 HD application.

Chapter 11

Further Research

The subject of this report is far more extensive than the scope of the project. Therefore, there is a lot of interesting possibilities for further research. Some of these aspects are described in this chapter.

As mentioned, MIKE 21 HD is a large and complex application, thus it was not possible to implement all the features of MIKE 21 HD in this project. An obvious next step would therefore be to add more of this complexity and investigate how it will affect the performance of the parallel application.

To have the opportunity to follow the water movements through all the intermediate time steps one will need to transfer data between host and device a lot of times. Because of the low bandwidth this would increase the execution time. However, using asynchronous transfers should make it possible to transfer all the intermediate time steps without increasing execution time. This means simply calculating next step while previous time steps is transferred back to the host memory.

In Section 9.4 it was shown how significant a performance increase could be obtained by switching from double- to single-precision. It would be interesting to further investigate how the decrease in precision influences the error of the application and if this could be minimized by applying mixed-precision iterative refinement.

The parallel CUDA C application is implemented such that the host (CPU) runs a `for`-loop over the time steps calling; two build, two solve and 14 transpose kernels each time step. The overhead from calling these kernels could be substantial. Therefore, it would be interesting to investigate how the application

will perform if only one kernel, execute the entire simulation. This means that calling both sweeps and loop over the time steps, will all be performed on the device (GPU). This will additionally make it possible to reduce global memory transfers by using shared memory to store the intermediate tri-diagonal system.

The hybrid CR-PCR algorithm showed significantly speedups compared to the other implemented solvers. However, this application could be further optimized as discussed in Section 8.4.2.

On the Fermi architecture it is possible to launch multiple kernels simultaneously. This implies that different systems could be simulated on the GPU simultaneously and in that way better utilize the hardware. It could therefore be interesting to investigate how fast multiple systems could be handled compared to MIKE 21 HD.

The two parallel approaches, developed in this project, have their different pros and cons. Therefore, combining these two approaches into a hybrid version and investigating how it would perform could be very interesting. This could involve building the system using one approach and solving it using the other.

When DHI realises how large systems can be simulated in a short amount of time, it would be reasonable to expect, that they want to simulate even larger systems. The two developed approaches are limited by the system size on the device memory, since all arrays are needed to be located on the device, which are very small compared to the host memory. Therefore, approaches where not all values are needed at once or where several GPUs are used to execute the application would be interesting to investigate further.

Appendix A

Nomenclature

Symbol	Decription	In this project
$h(x,y,t)$	water depth, i.e. $h = \zeta - d$, $[m]$	
$d(x,y,t)$	ground surface elevation (bathymetry), $[m]$	is constant
$\zeta(x,y,t)$	water surface elevation, $[m]$	
$p(x,y,t)$,$q(x,y,t)$	flux densities in x- and y-directions, $[m^3/s/m]$. Basically fluid velocity in the given direction times water surface elevation	
$C(x,y)$	Chezy resistance, $[m^{1/2}/s]$	
M	Manning coefficient, unit less	set to 32
g	acceleration due to gravity, $[m/s^2]$	9.81
$f(V)$	wind friction factor	set to zero
$V, V_x, V_y(x,y,t)$	wind speed and components in x- and y-directions, $[m/s]$	set to zero

$\Omega(x,y)$	Coriolis parameter, latitude dependent, $[s^{-1}]$	set to zero
$p_a(x,y,t)$	atmospheric pressure, $[kg/m/s^2]$	set to constant
$\tau_{xx}, \tau_{xy}, \tau_{yy}$	components of effective shear stress	set to constant
x, y	Cartesian coordinates, $[m]$	
t	time, s	
Δx, Δy	grid distance in x- and y-direction, $[m]$	
j, k	index in x- and y-direction, thus j is the j^{th} column and k is the k^{th} row	

Table A.1: Nomenclature

Appendix B

Platforms Specification

Here are listed specifications of the used test environments. The key values are taking from the platforms, `deviceQuery` from NVIDIA CUDA SDK C/C++ and from [29, tab. F-2], where further Fermi architecture specifications can be found.

Test Environment

CPU	Intel(R) Xeon(R) E5620 @ 2.40GHz
Cache size	12288 kB
Cores	4
Max Memory Bandwidth	25.6 GB/s
RAM	12 GB
GPU	NVIDIA GeForce GTX 590 (2x GF110 chip)
Operating system	Ubuntu 10.04.4 LTS (x86_64 GNU/Linux)
CUDA Driver/Runtime	Version 4.1/4.1
NVIDIA Visual Profiler	Version 4.1
CUDA Occupancy Calculator	Version 2.4

Table B.1: Specifications of test environment.

NVIDIA GeForce GTX 590

Chip	2x GF110
Compute Capability	2.0
RAM (GDDR5)	1,536 MB
Multiprocessors	16
CUDA Cores	512
GPU Clock Speed	1.22 GHz
Memory Clock rate	1707.00 Mhz
Memory Bus Width	384-bit
Memory bandwidth	163.87 GB/s
Performance(single-precision)	1244.15 Gflops
Support host page-locked memory mapping	Yes
Device has ECC support enabled	No
L2 Cache Size	768 kB
Maximum amount of shared memory per multiprocessor	48 kB
Number of 32-bit registers per multiprocessor	32,768
Maximum dimensionality of grid of thread blocks	3
Maximum dimensionality of thread block	3
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535
Maximum x- or y-dimension of a block	1024
Warp size	32
Maximum number of resident blocks per multiprocessor	8
Maximum number of resident warps per multiprocessor	48
Maximum number of resident threads per multiprocessor	1536
Number of shared memory banks	32

Table B.2: Specifications of GPU NVIDIA GeForce GTX 590 used in tests.

DHI Test Environment

CPU	Intel(R) Core(TM) i3-2120 @ 3.30 GHz
Cache size	3072 kB
Cores	2
Max Memory Bandwidth	21 GB/s
RAM	8 GB
Operating system	Windows 7 (64-bit)
Profiler	AQtime Standard, Version 7.40.800.647

Table B.3: Specifications of DHI test environment.

Appendix C

Project Description Provided by DHI

Here are given the original project description provided by DHI.

WATER WAVE DYNAMICS: 2D FLOW EQUATIONS ON A GPU

DHI uses a 2D free-surface flow numerical engine to simulate water movement in lakes, estuaries, bays, coastal areas and seas, based on rain, tidal variation, wind etc. Applications include prediction of tidal hydraulics, wind and wave generated currents, storm surges, waves in harbours, dam-breaks and tsunamis. The DHI product used for these simulations is called MIKE 21.

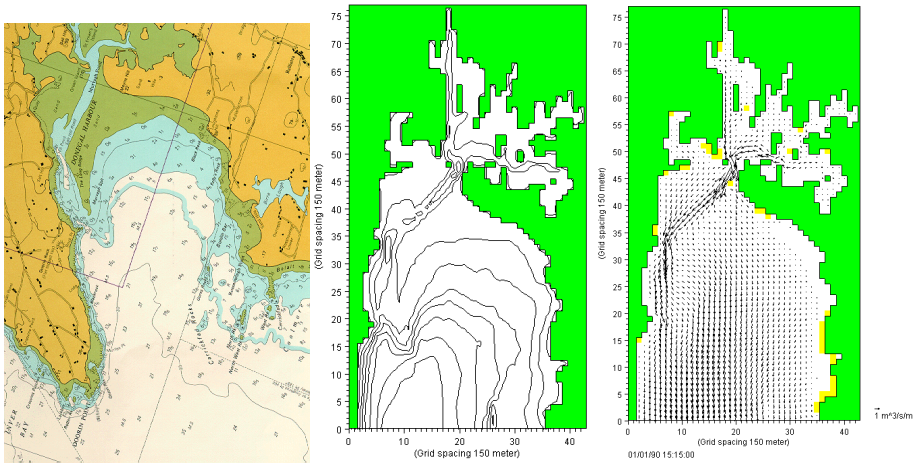


Figure: Map, bathymetry and flow field from a MIKE 21 model

Project

MIKE 21 solves the shallow water equation on a rectangular grid, using a finite difference method. The solution scheme is ADI (Alternating Direction Implicit), which solves the equations for one column or row at a time, in an alternating order.

The goal of the project is to formulate a parallel solution scheme for MIKE 21 and implement it on a GPU. The expected outcome is an improvement in simulation speed.

The impact will be significant, especially for optimization settings where it is necessary to run several hundred simulations. A drastic improvement in simulation speed has the potential to change the type of optimization problems where MIKE 21 is applicable and thereby open new market segments for DHI.

Prerequisites

DHI will provide

1) a detailed problem description,
2) the shallow water equations, the discretization scheme and the current solution algorithm,
3) the existing FORTRAN implementation,
4) a skeleton code in c with data structures that follow the FORTRAN implementation.

Based on the skeleton c code, a parallel solution must be implemented in c and converted to CUDA. At the end of the project it should be possible to incorporate the c/CUDA code into the existing MIKE 21 FORTRAN code.

Therefore, it is required that the student has c and CUDA skills. However, knowledge of water wave dynamics or FORTRAN is not required.

Appendix D

Bandwidth Test

Here are listed results from executing the `bandwidthTest` provided by NVIDIA CUDA C/C++ SDK Code Samples, with pageable and pinned memory allocation.

Listing D.1: Execution of `bandwidthTest`

```
s093053@gpulab04:/usr/local/cuda-4.1/sdk/C/bin/linux/release$ ./
    bandwidthTest --memory=pageable
[bandwidthTest] starting...

./bandwidthTest Starting...

Running on...

 Device 0: GeForce GTX 590
 Quick Mode

 Host to Device Bandwidth, 1 Device(s), Paged memory
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     5310.4

 Device to Host Bandwidth, 1 Device(s), Paged memory
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     4218.4

 Device to Device Bandwidth, 1 Device(s)
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     132865.6

[bandwidthTest] test results...
PASSED

> exiting in 3 seconds: 3...2...1...done!
```

```
27
28 s093053@gpulab04:/usr/local/cuda-4.1/sdk/C/bin/linux/release$ ./
       bandwidthTest --memory=pinned
29 [bandwidthTest] starting...
30
31 ./bandwidthTest Starting...
32
33 Running on...
34
35  Device 0: GeForce GTX 590
36  Quick Mode
37
38  Host to Device Bandwidth, 1 Device(s), Pinned memory
39    Transfer Size (Bytes)        Bandwidth(MB/s)
40    33554432                     5797.3
41
42  Device to Host Bandwidth, 1 Device(s), Pinned memory
43    Transfer Size (Bytes)        Bandwidth(MB/s)
44    33554432                     6237.4
45
46  Device to Device Bandwidth, 1 Device(s)
47    Transfer Size (Bytes)        Bandwidth(MB/s)
48    33554432                     132841.8
49
50 [bandwidthTest] test results...
51 PASSED
52
53 > exiting in 3 seconds: 3...2...1...done!
```

Appendix E

Source Code to Performance Test

Here are listed the source code, which is used when testingthe peak computation rate in single- and double-precision. The code test a lot of multiply-add operations.

Listing E.1: Source code used when testing performance

```
1  /*
2  * Copyright 1993-2007 NVIDIA Corporation.  All rights reserved.
3  *
4  * NOTICE TO USER:
5  *
6  * This source code is subject to NVIDIA ownership rights under U.S. and
7  * international Copyright laws.  Users and possessors of this source
       code
8  * are hereby granted a nonexclusive, royalty-free license to use this
       code
9  * in individual and commercial software.
10 *
11 * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
12 * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
13 * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
14 * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
15 * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR
       PURPOSE.
16 * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT,
       INCIDENTAL,
17 * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM
       LOSS
18 * OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT,
       NEGLIGENCE
19 * OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE
```

```
    USE
20  * OR PERFORMANCE OF THIS SOURCE CODE.
21  *
22  * U.S. Government End Users.   This source code is a "commercial item"
        as
23  * that term is defined at  48 C.F.R. 2.101 (OCT 1995), consisting  of
24  * "commercial computer  software"  and "commercial computer software
25  * documentation" as such terms are  used in 48 C.F.R. 12.212 (SEPT
        1995)
26  * and is provided to the U.S. Government only as a commercial end item.
27  * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
28  * 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
29  * source code with only those rights set forth herein.
30  *
31  * Any use of this source code in individual and commercial software
        must
32  * include, in the user documentation and internal comments to the code,
33  * the above Disclaimer and U.S. Government End Users Notice.
34  */
35
36  /*
37     This sample is intended to measure the peak computation rate of the
           GPU in GFLOPs
38     (giga floating point operations per second).
39
40     It executes a large number of multiply-add operations, writing the
           results to
41     shared memory. The loop is unrolled for maximum performance.
42
43     Depending on the compiler and hardware it might not take advantage
           of all the
44     computational resources of the GPU, so treat the results produced by
            this code
45     with some caution.
46  */
47
48  #include <stdlib.h>
49  #include <stdio.h>
50  #include <string.h>
51  #include <math.h>
52
53  #include <cutil.h>
54
55  #define NUM_SMS (24)
56  #define NUM_THREADS_PER_SM (384)
57  #define NUM_THREADS_PER_BLOCK (192)
58  #define NUM_BLOCKS ((NUM_THREADS_PER_SM / NUM_THREADS_PER_BLOCK) *
        NUM_SMS)
59  #define NUM_ITERATIONS 32
60
61  // 128 MAD instructions
62  #define FMAD128(a, b) \
63       a = b * a + b; \
64       b = a * b + a; \
65       a = b * a + b; \
```

```
66      b = a * b + a; \
67      a = b * a + b; \
68      b = a * b + a; \
69      a = b * a + b; \
70      b = a * b + a; \
71      a = b * a + b; \
72      b = a * b + a; \
73      a = b * a + b; \
74      b = a * b + a; \
75      a = b * a + b; \
76      b = a * b + a; \
77      a = b * a + b; \
78      b = a * b + a; \
79      a = b * a + b; \
80      b = a * b + a; \
81      a = b * a + b; \
82      b = a * b + a; \
83      a = b * a + b; \
84      b = a * b + a; \
85      a = b * a + b; \
86      b = a * b + a; \
87      a = b * a + b; \
88      b = a * b + a; \
89      a = b * a + b; \
90      b = a * b + a; \
91      a = b * a + b; \
92      b = a * b + a; \
93      a = b * a + b; \
94      b = a * b + a; \
95      a = b * a + b; \
96      b = a * b + a; \
97      a = b * a + b; \
98      b = a * b + a; \
99      a = b * a + b; \
100     b = a * b + a; \
101     a = b * a + b; \
102     b = a * b + a; \
103     a = b * a + b; \
104     b = a * b + a; \
105     a = b * a + b; \
106     b = a * b + a; \
107     a = b * a + b; \
108     b = a * b + a; \
109     a = b * a + b; \
110     b = a * b + a; \
111     a = b * a + b; \
112     b = a * b + a; \
113     a = b * a + b; \
114     b = a * b + a; \
115     a = b * a + b; \
116     b = a * b + a; \
117     a = b * a + b; \
118     b = a * b + a; \
119     a = b * a + b; \
120     b = a * b + a; \
```

```
121     a = b * a + b; \
122     b = a * b + a; \
123     a = b * a + b; \
124     b = a * b + a; \
125     a = b * a + b; \
126     b = a * b + a; \
127     a = b * a + b; \
128     b = a * b + a; \
129     a = b * a + b; \
130     b = a * b + a; \
131     a = b * a + b; \
132     b = a * b + a; \
133     a = b * a + b; \
134     b = a * b + a; \
135     a = b * a + b; \
136     b = a * b + a; \
137     a = b * a + b; \
138     b = a * b + a; \
139     a = b * a + b; \
140     b = a * b + a; \
141     a = b * a + b; \
142     b = a * b + a; \
143     a = b * a + b; \
144     b = a * b + a; \
145     a = b * a + b; \
146     b = a * b + a; \
147     a = b * a + b; \
148     b = a * b + a; \
149     a = b * a + b; \
150     b = a * b + a; \
151     a = b * a + b; \
152     b = a * b + a; \
153     a = b * a + b; \
154     b = a * b + a; \
155     a = b * a + b; \
156     b = a * b + a; \
157     a = b * a + b; \
158     b = a * b + a; \
159     a = b * a + b; \
160     b = a * b + a; \
161     a = b * a + b; \
162     b = a * b + a; \
163     a = b * a + b; \
164     b = a * b + a; \
165     a = b * a + b; \
166     b = a * b + a; \
167     a = b * a + b; \
168     b = a * b + a; \
169     a = b * a + b; \
170     b = a * b + a; \
171     a = b * a + b; \
172     b = a * b + a; \
173     a = b * a + b; \
174     b = a * b + a; \
175     a = b * a + b; \
```

```
176       b = a * b + a; \
177       a = b * a + b; \
178       b = a * b + a; \
179       a = b * a + b; \
180       b = a * b + a; \
181       a = b * a + b; \
182       b = a * b + a; \
183       a = b * a + b; \
184       b = a * b + a; \
185       a = b * a + b; \
186       b = a * b + a; \
187       a = b * a + b; \
188       b = a * b + a; \
189       a = b * a + b; \
190       b = a * b + a; \
191
192 __shared__ float result[NUM_THREADS_PER_BLOCK];
193
194 __global__ void gflops()
195 {
196     float a = result[threadIdx.x];  // this ensures the mads don't get
            compiled out
197     float b = 1.01f;
198
199     for (int i = 0; i < NUM_ITERATIONS; i++)
200     {
201         FMAD128(a, b);
202         FMAD128(a, b);
203         FMAD128(a, b);
204         FMAD128(a, b);
205         FMAD128(a, b);
206         FMAD128(a, b);
207         FMAD128(a, b);
208         FMAD128(a, b);
209         FMAD128(a, b);
210         FMAD128(a, b);
211         FMAD128(a, b);
212         FMAD128(a, b);
213         FMAD128(a, b);
214         FMAD128(a, b);
215         FMAD128(a, b);
216         FMAD128(a, b);
217     }
218     result[threadIdx.x] = a + b;
219 }
220
221 int
222 main(int argc, char** argv)
223 {
224     CUT_DEVICE_INIT(argc,argv);
225
226     // warmup
227     gflops<<<NUM_BLOCKS, NUM_THREADS_PER_BLOCK>>>();
228     CUDA_SAFE_CALL( cudaThreadSynchronize() );
229
```

```
230     // execute kernel
231     unsigned int timer = 0;
232     CUT_SAFE_CALL( cutCreateTimer( &timer));
233     CUT_SAFE_CALL( cutStartTimer( timer));
234 
235     gflops<<<NUM_BLOCKS, NUM_THREADS_PER_BLOCK>>>();
236 
237     CUDA_SAFE_CALL( cudaThreadSynchronize() );
238     CUT_SAFE_CALL( cutStopTimer( timer));
239     float time = cutGetTimerValue( timer);
240 
241     // output results
242     printf( "Time: %f (ms)\n", time);
243     const int flops = 128 * 2 * 16 * NUM_ITERATIONS * NUM_BLOCKS *
            NUM_THREADS_PER_BLOCK;
244     printf("Gflops: %f\n", (flops / (time / 1000.0f)) / 1e9 );
245 
246     CUT_SAFE_CALL( cutDeleteTimer( timer));
247     CUT_EXIT(argc, argv);
248 }
```

Bibliography

[1] S. Allmann, T. Rauber, and G. Runger. Cyclic reduction on distributed shared memory machines. In *Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop on*, pages 290 –297, 2001. doi: 10.1109/EMPDP.2001.905055.

[2] Angelini, Chris; tom's hardware. GeForce GTX 580 And GF110: The Way Nvidia Menat It To Be Played, 2010. http://www.tomshardware.com/reviews/geforce-gtx-580-gf110-geforce-gtx-480,2781-2.html. Last visited on July 2, 2012.

[3] Angelini, Chris; tom's hardware. Benchmark Results: Sandra 2012, 2012. http://www.tomshardware.com/reviews/geforce-gtx-680-review-benchmark,3161-14.html. Last visited on July 25, 2012.

[4] Assistant Professor Allan P. Engsig-Karup. Welcome to 02685 Scientific Computing for Differential Equations. Slide from lecture 1 Introduction.

[5] NVIDIA Corporation. Compute command line profiler. DU-05982-001_v03, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Compute_Command_Line_Profiler_User_Guide.pdf.

[6] NVIDIA Corporation. Compute visual profiler. DU-05162-001_v04, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Compute_Visual_Profiler_User_Guide.pdf.

[7] NVIDIA Corporation. The cuda compiler driver nvcc, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/nvcc.pdf.

[8] Andrew Davidson and John D. Owens. Register packing for cyclic reduction: a case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 4:1–4:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0569-3. doi:

10.1145/1964179.1964185. URL http://doi.acm.org.globalproxy.cvt.dk/10.1145/1964179.1964185.

[9] DHI Group. http://www.dhigroup.com/, http://www.mikebydhi.com/Products/CoastAndSea/MIKE21/Hydrodynamics.aspx. Last visited on July 25, 2012.

[10] Dr. Dobbs; The World of Software Development.

[11] EPA; United States Environmental Protection Agency. Global Climate Change - Coastel Areas, 2011. http://epa.gov/climatechange/kids/impacts/effects/coastal.html. Last visited on March 14, 2012.

[12] Rob Farber. *CUDA Application Design and Development*. Morgan Kaufmann. Elsevier Science, 2011. ISBN 9780123884268. URL http://books.google.dk/books?id=MtLvlQvYDOEC.

[13] D. Goddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on gpus applied to mixed-precision multigrid. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):22 –32, January 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.61.

[14] W.M.W. Hwu. *Gpu Computing Gems Jade Edition*. Morgan Kaufmann's Applications of Gpu Computing. Elsevier Science, 2011. ISBN 9780123859631. URL http://www.google.dk/books?id=dNvantWW7HMC.

[15] Hee-Seok Kim, Shengzhao Wu, Li wen Chang, and W.W. Hwu. A scalable tridiagonal solver for gpus. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 444–453, sept. 2011. doi: 10.1109/ICPP.2011.41.

[16] R.J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, 2007. ISBN 9780898716290. URL http://books.google.dk/books?id=Cbx7QgAACAAJ.

[17] Micikevicius, Paulius; NVIDIA Corporation. Analysis-Driven Optimization, 2010. http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Analysis_Driven_Optimization.pdf.

[18] Micikevicius, Paulius; NVIDIA Corporation. Local Memory and Register Spilling, 2011. http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

[19] Micikevicius, Paulius; NVIDIA Corporation. Identifying Performance Limiters, 2011. http://developer.download.nvidia.com/CUDA/training/cuda_webinars_dentifying_performance_limiters.pdf.

[20] Micikevicius, Paulius; NVIDIA Corporation. Performance Optimization, 2011. http://www.nvidia.com/docs/IO/116711/sc11-perf-optimization.pdf.

[21] MIKE By DHI. MIKE 21 FLOW MODEL Hydrodynamic Module, 2009. Internal Scientific Documentation of MIKE 21 HD.

[22] NVIDIA. INTRODUCTING CUDA 5, . http://www.nvidia.com/object/cuda_home_new.html. Last visited on July 25, 2012.

[23] NVIDIA. What is GPU Computing?, . http://www.nvidia.com/object/GPU_Computing.html. Last visited on July 25, 2012.

[24] NVIDIA, . http://developer.nvidia.com/. Last visited on July 2, 2012.

[25] NVIDIA. TESLA PERSSONAL SUPERCOMPUTING WORKSTATION SOLUTIONS, . http://www.nvidia.com/object/personal-supercomputing.html. Last visited on July 26, 2012.

[26] NVIDIA. FERMI COMPATIBILITY GUIDE FOR CUDA APPLICATIONS. DA-05607-001_v1.5, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Fermi_Compatibility_Guide.pdf.

[27] NVIDIA Corporation. CUDA GPU Occupancy Calculator, v2.4. http://developer.download.nvidia.com/compute/cuda/4_0/sdk/docs/CUDA_Occupancy_Calculator.xls.

[28] NVIDIA Corporation. PTX: Parallel Thread Execution, ISA v2.3, 2009. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf.

[29] NVIDIA Corporation. NVIDIA CUDA C Programming Guide v4.1, 2011.

[30] NVIDIA Corporation. USING INLINE PTX ASSEMBLY IN CUDA. DA-05713-001_V01, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Using_Inline_PTX_Assembly_In_CUDA.pdf.

[31] NVIDIA Corporation. TUNING CUDA APPLICATIONS FOR FERMI. DA-05612-001_v1.5, 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Fermi_Tuning_Guide.pdf.

[32] NVIDIA Corporation. CUDA C BEST PRACTICES GUIDE. DG-05603-001_v4.1, 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf.

[33] NVIDIA CUDA. NVIDIAs Next Generation CUDA Compute Architecture Fermi, Whitepaper v1.1, . http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[34] NVIDIA CUDA. NVIDIAs Next Generation CUDA Compute Architecture Kepler GK110 v1.0, . http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[35] NVIDIA GeForce. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications.

[36] NVIDIA Library Documentation. NVIDIA CUDA Library Documentation 4.2. http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/docs/online/index.html.

[37] Volkov, Vasily. Better Performance at Lower Occupancy, 2010. http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[38] Wang, Peng; Developer Technology NVIDIA. Analysis-Driven Optimizations in CUDA. http://developer.download.nvidia.com/GTC/PDF/1082_Wang.pdf.

[39] Whitehead, Nathan and Fit-Florea, Alex. Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.

[40] Wikipedia; The Free Encyclopedia. Alternating direction implicit method, 2011. http://en.wikipedia.org/wiki/Alternating_direction_implicit_method. Last visited on July 25, 2012.

[41] Wikipedia; The Free Encyclopedia. Nvidia Tesla, 2012. http://en.wikipedia.org/wiki/Nvidia_Tesla. Last visited on July 25, 2012.

[42] Wikipedia; The Free Encyclopedia. Shallow water equations, 2012. http://en.wikipedia.org/wiki/Shallow_water_equations. Last visited on July 25, 2012.

[43] Wikipedia; The Free Encyclopedia. 2004 Indian Ocean earthquake and tsunami, 2012. http://en.wikipedia.org/wiki/2004_Indian_Ocean_earthquake_and_tsunami. Last visited on March 14, 2012.

[44] Wikipedia; The Free Encyclopedia. CUDA, 2012. http://en.wikipedia.org/wiki/CUDA. Last visited on July 22, 2012.

[45] Wikipedia; The Free Encyclopedia. Comparison of Nvidia graphics processing units, 2012. http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units. Last visited on July 25, 2012.

[46] Wolfe, Michael. The Heterogeneous Programming Jungle, 2012. http://www.hpcwire.com/hpcwire/2012-03-19/the_heterogeneous_programming_jungle.html. Last visited on July 25, 2012.

[47] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the gpu. *SIGPLAN Not.*, 45(5):127–136, January 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693472. URL http://doi.acm.org.globalproxy.cvt.dk/10.1145/1837853.1693472.

[48] Ziegler, Gernot; NVIDIA Corporation. Analysis-Driven Optimization, ISC 2011 Tutorial, 2011. http://www.nvidia.com/content/PDF/isc-2011/Ziegler.pdf.